Kakos_nonos

СТЕПЛЕР

Язык программирования

Содержание

1.1 Вступление с отступлением

Здравствуйте, уважаемые читатели! Сейчас вы держите в руках (или читаете на компьютере) книгу об очень интересном языке программирования – степлере. Почему мы будем говорить именно о нём, о степлере, а не о каком-то другом языке программирования? На это есть много причин:

Во-первых, любой язык программирования имеет право на существование, будь то очень популярный C++, или никому не известный p'', неважно, этот язык существует, и это прекрасно.

Во-вторых, степлер - своеобразный язык. В нём есть несколько достаточно интересных особенностей, которых нет в других языках программирования: начиная с самой идеи – минимальный язык с максимальными возможностями – ни у какого другого языка нет такой идеи, и заканчивая тем, что его нельзя отнести ни к одной из ныне существующих парадигм.

В-третьих, изучать степлер можно и для общего развития. Во время прочтения книги вы можете узнать для себя что-то новое, повторить знания по программированию, а также, что немаловажно, увеличить количество нервных клеток, что приведёт к улучшению интеллекта.

В-четвёртых, это просто интересно.

Как видите, есть несколько немаловажных причин, чтобы начать изучать этот интересный язык. Поэтому давайте не будем терять время и начнём это увлекательное занятие.

1.2 Об языке

Здесь мы постараемся объяснить, что такое степлер, и где он может быть использован.

Начнём своё повествование с краткого экскурса в историю: Степлер был придуман достаточно недавно, а именно 14 февраля 2011 года, примерно в 9-10 часов вечера. При создании ставилась цель создать минимальный язык с максимальными возможностями. Правда, сейчас минимальным его уже достаточно трудно назвать, но всё равно, количество его команд намного меньше, чем в других языках, а по возможностям он им ничем ни уступает. Так вот, сейчас уже есть третья версия языка, которая во много раз мощнее первой и содержит больше возможностей, а до этого была вторая, которая была слабее, чем третья, и первая, которая описана в моей прошлой книге.

Теперь давайте поговорим о самом степлере как о языке программирования. Для начала, надо определить, к какой парадигме он относится. И вот тут определяется, что этого сделать нельзя, так как степлер просто не подходит ни под одну известную парадигму, в нём собраны все известные на данный момент парадигмы. Также его можно назвать эзотерическим, но уж слишком он мощный для эзотерического языка.

Если говорить о том, является ли язык компилируемым или интерпретируемым, то язык является более интерпретируемым, потому что в нём есть несколько аспектов самомодификации, а с ними попробуй напиши компилятор – трудновато будет, но, возможно, потому что оно так и есть.

Чтобы не быть голословным, приведу пример простой программки:

```
Program
[Prime number Founder 2.0]
[From]
$(6)(2$^1-)
[To]
$(7)(2$)
{next}
$(6)(6$^1+)
$(5)(6$@)
$(4)(1)
{nl}
$(4)(4$^1+)
#(6$^4$%)<l>
#(4$^5$-|)<nl>
$(2)(6$)
{l}
#(6$^7$-|)<next>
```

Эта программа выполняет поиск простых чисел во введённом диапазоне. Как видите, непосвящённым ничего не понятно - в этом и есть великая мощь степлера!

1.3 Особенности Степлера

Кроме своих обычных назначений, у степлера открылось несколько весьма интересных особенностей.

Одна из них наиболее примечательна: когда человек программирует на степлере, у него происходит сильный приток крови к голове, что вызывает активное нарастание нервных клеток. Это спровоцировано тем, что программирование на степлере кардинально отличается от программирования на других языках . Благодаря этому мозг, при переходе с другого языка на степлер, меняет свою структуру, и в это время нервные клетки усиленно растут. После того, как вы достаточно много программировали на степлере и ваш мозг перестроился, нужно перейти на другой язык и немного покодить на нём.

Другой язык программирования может быть любым, поскольку степлер серьёзно отличается от всех их, вместе взятых.

Внимание!!! Занятие степлером может иметь и плохие последствия. Например, если вы занимаетесь степлером слишком много, то ваш интеллект будет расти настолько сильно, что может появится суперразум. Для кого-то это и хорошо, а для кого-то не очень. Так что будьте осторожны и бдительны.

1.4 Спасибы

Я хотел бы в первую очередь сказать спасибо себе, за то что сделал этот язык и написал эту книгу. Также хочу выразить благодарность Абадяберу за то, что он написал свой Интерпретатор, компилятор и вообще, сделал очень много для развития языка; Admin'y с iForum'a за то, что сделал этот форум, моему коту Рыжику за то, что давал себя

гладить, собаке Топику за то, что охранял, Биллу Гейтсу за DOS, а также всем остальным, кому за что.

Отдельное спасибо хочу сказать буквам русского алфавита, без чьей помощи эта книга не была бы написана.

1.5 Контакты

Разумеется, во время чтения книги у вас могут возникнуть различные вопросы по языку степлеру или по чему-либо другому. Вначале я советую самому попытаться разобраться с этим вопросом, вдруг получится, а потом уже, если не получается, написать мне или куда-нибудь ещё.

Мне можно написать вот такими образами:

Моё мыло Kakos_nonos@mail.ru туда вы можете написать свой вопрос, и я вам отвечу.

Если вы состоите в базе данных ФСБ Вконтакте, то можете написать на Vk.com/kakos_nonos .

Также можете найти меня в джаббере: Kakos_nonos@jabber.ru, а вот мой сайт: kabardcomp.narod.ru

Существует также несколько групп, посвящённых степлеру. На форуе iforum.su есть раздел о степлере. Вот он: http://iforum.su/stepler-76 и группа: http://iforum.su/groups/3-stepler.html

Можно также обратится к ещё одному специалисту в этой области – Абадяберу: abaduaber.narod.ru

1.6 Структура книги

Эта книга состоит из трёх частей. В первой рассматриваются команды языка, во второй - алгоритмы,

потому что, зная одни лишь команды, ни шута не напишешь, и, наконец, третья часть, в которой рассматриваются примеры программ.

Самое большое внимание было уделено второй части, потому что команды это команды, они только описывают язык, примеры тоже хорошо, из них можно выудить уйму информации, да вот только кто это будет делать. Поэтому самую большую часть книги занимают именно алгоритмы. Вот так.

В книге есть несколько обозначений:

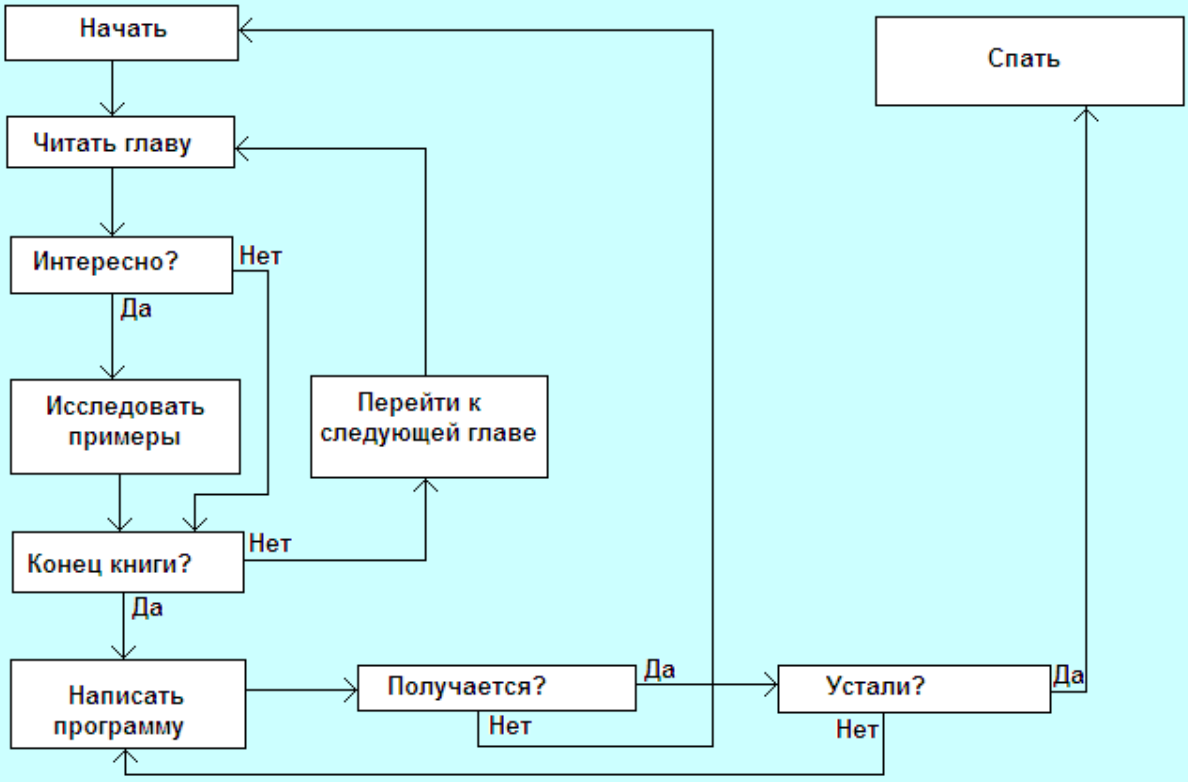
Вопросик. Надо будет немного подумать. Что поделать, везде надо подумать, ведь это так важно. Но, в общем, не так.

Какая-то опасность.

Использование на практике, изучение, и, в особо тяжёлых случаях, даже внимательное чтение может очень серьёзно повлиять на вас и на ваш компьютер. Так что – сами виноваты.

Для большего удобства была разработана схема чтения этой книги. Если вы будете ей следовать, то

чтение этой книги станет очень увлекательным занятием.



Наконец, можно приступить непосредственно к изучению языка.

2.1 Первая программа

Сейчас мы напишем простую программу. Эта программа будет просто выводить на экран обычную текстовую строчку с глупым текстом. А почему не ХеллоуВорлд? – спросите вы. Да потому что всякие там ХеллоуВорлд применяются в обычных языках, а степлер необычный язык.

Так вот сама программа:

```
Program
[Stupid Text].
```

Попытаемся разобраться с ней поподробнее. Вначале идёт команда Program. Она нужна во всех программах и

указывает на начало кода, или, как говорят кульхацкеры, точку входа.

Далее, в квадратных скобках, идёт сам текст, который будет выводиться на экран. Этот текст не должен содержать закрывающей квадратной скобки, а то поди разберись, или это скобка, или это конец строки.

Ещё в степлере есть комментарий. Его можно писать после точки с запятой, вот так:

```
[TROLOLO] ;Выводим на экран надпись
```

2.2 Команды пересылки

Команда пересылки данных или присваивания является основной в стелпере. Все остальные команды базируются на ней, и если вы её не поймёте, дальше можно будет не читать. Конечно, можно будет прочитать, но вот только зачем?

Перед изучением команды присваивания надо разобраться с системой хранения переменных.

Она очень проста: есть один массив, в котором хранятся все переменные. Доступ к ним осуществляется по их номеру, то есть, вместо имён у переменных есть номера. Потом мы научимся давать переменных имена, но об этом попозже. Каждая переменная может принимать значения от -32767 и до 32768. Ещё надо помнить о том, что первые две переменные зарезервированы для ввода/вывода, о них мы будем говорить в разделе ввод/вывод, а пока надо просто помнить о том, что их нельзя использовать для хранения данных.

Команда присваивания имеет вот такой синтаксис:

```
$(приёмник)(источник)
```

Вначале идёт знак доллара, потом, в скобках, номер переменной, в которую мы будем запихивать данные, а потом, во второй скобке, находятся данные, которые мы туда засовываем.

Вот пример команды. После её выполнения ячейка 15 будет равна 168:

$(15)(168)

А как присвоить одной переменной значение другой переменной? А очень просто. Надо после номера числа поставить знак доллара, и тогда это будет уже не просто число, а будет переменная, именно число, которое лежит в этой переменной. Немного непонятно, не правда ли? Разберёмся на примерах:

17 – это число 17. 17$ - это 17-ая переменная.

$(5)(4$) - копировать в 5-ую переменную значение 4-ой переменной.

$(4$)(8$) – копировать значение переменной 8 в переменную, номер которой находится в 4-ой переменной.

Также степлер может работать и с символами. Вот небольшой пример:

$(5)('d') – копировать в ячейку 5 ASCII-код буквы d.

Подумайте, что означает команда $(5)(7$$)?

2.3 Ввод/вывод

Сейчас мы научимся выводить значения на экран, а то какая польза от того, что там лежит в наших переменных. Для этого зарезервировано две переменные – первая и

вторая. Чтение и запись в них будет сопровождаться выводом на экран и вводом с клавиатуры значений.

Первая переменная служит для ввода/вывода символьных значений, то есть при записи в неё на экране появится символ, ASCII-код которого мы туда записали. А если же попытаться прочитать значение из первой ячейки, то программа будет ожидать нажатия клавиши и выдаст код этой клавиши.

Со второй переменной дела обстоят почти также, только вместо символов будут числа.

Вот несколько примеров:

$(2)(123) – вывести на экран число 123

$(1)(100) – напечатать на экране букву d(код 100)

$(1)('u') – напечатать на экране букву u

$(5)(2$) – прочитать число с клавиатуры, и положить его в пятую ячейку.

$(7)(1$) – прочитать символ с клавиатуры и его код сохранить в 7-ой ячейке.

В итоге, у нас готова карта памяти степлера. Вот она, на рисунке 00h.

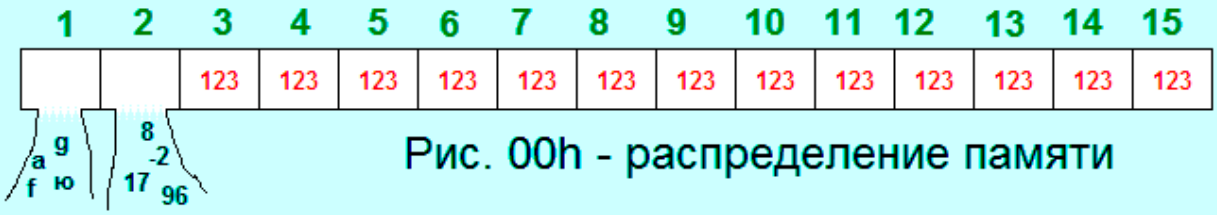


Рис. 00h - распределение памяти

Подумайте, что значит команда $(2)(1$)?

2.4 Запись числа

Вроде бы команда присваивания достаточно примитивная и с её помощью можно мало чего интересного сделать. Однако, это не так. С её помощью можно делать сложные вычисления и тому подобное. Как? – спросите вы. Очень легко. Дело в том, что в степлере числом является намного более сложная структура, чем просто набор цифр.

Числом (или, если сказать более чётко, записью числа) в степлере является выражение на обратной польской нотации. Если во время прочтения этой фразы вы подумали о Польше, то рекомендуем немного отдохнуть и почитать википедию. Эта запись состоит из односимвольных операторов, которые выполняют арифметические, логические и другие операции над числами. Обратная польская запись использует стек. Мы будем называть его стек обратной польской записи (стек ОПЗ). Всего степлер использует 4 (или три, смотря как считать) стека.

Сейчас мы будем изучать эти операторы.

AAAAAAAAa!

Если во время чтения этой главы вы почувствовали шум в ушах, головную боль или галлюцинации, настоятельно рекомендуем вам воздержаться от дальнейшего чтения и заняться любимым делом, например, бизнесом. Если что не так – я не виноват.

2.4.1 Системы счисления

Степлер поддерживает аж три системы счисления: Десятичную (с которой вы уже знакомы), двоичную и шестнадцатеричную.

Десятичная система обозначается просто: если нет идентификатора, то это число в десятичной системе счисления. Примеры, думаю, приводить не стоит, и так понятно.

Числа в шестнадцатеричной системе счисления пишутся так: ~h07B , то есть, вначале пишется загогулина (~), потом буковка h (или H), а потом уже само шестнадцатеричное число (тут 076, то есть в десятичной системе счисления 118).

Двоичная система счисления описывается также, только вместо h там b. Вот пример: ~b010010

Числа в шестнадцатеричной и двоичной системах счисления можно использовать также, как и в десятичной: $(~b10)(~h54$)

2.4.2 Арифметика

Сейчас мы немного займёмся математикой. Да-да, в степлере будет много математики, и если вы её не знаете, советуем вам почитать её на уровне 4-ого класса.

Перед тем, как подойти к изучению математических операторов, следует изучить один очень важный оператор – вверх. В степлере он обозначается вот таким символом:^. Кто-то называет его крышей, кто-то домиком, а вообще он называется циркумфлекс. Он делает вот что: перемещает все значения в стеке на один уровень вверх, а вершину стека ОПЗ приравнивает к нулю. Например, вот

значения стека до операции: 3, 4, 5, 10. А вот после: 3, 4, 5, 10, 0.

Далее, когда мы подняли значения на один уровень вверх, мы можем заполнить его вершину другим числом, например: 53^41 после этих операторов вершина стека будет равна 41, а подвершина – 53. Вот так.

Теперь можно приступать к изучению математических операций. Математические операции работают так: берут из стека числа, выполняют вычисления и возвращают туда же результат.

Теперь можно изучать уже сами операторы. Вот они, родимые.

+ - Складывает вершину и подвершину.

- - Отнимает от подвершины вершину.

* - Умножает подвершину на вершину.

/ - Делит подвершину на вершину.

@ - Вычисляет квадратный корень из вершины.

& - Возводит вершину в степень подвершины.

% - Получает остаток от деления подвершины на вершину.

Рассмотрим несколько примеров, как это использовать. Вообще принцип простой: 2^2+ то есть мы пишем число, поднимаем его на один уровень вверх, и потом выполняем операцию. То есть, это выражение будет значить то же, что и 2+2.

17^8- 17-8

2^8* 2*8

2^9& 9 во второй степени

4@ корень из 4

Также можно делать цепочки из операций, тогда получается вот такие вещи:

24^7+^8*^2^3++

Это работает так: Вначале считается сумма 24 и 7, потом это умножается на 8 и потом в стек поднимаются 2 и 3, и это всё прибавляются к тому произведению.

Да, это немного трудно, но потом будет ещё труднее, поэтому подготовьтесь. На самом деле, надо просто разобраться с обратной польской нотацией и стеком, тогда всё будет выглядеть просто.

А если скомбинировать это с тем, что мы уже знаем, то получится вообще супер:

$(~h47^~b101$*)(5^~h22$+^8-)

На самом деле, это тоже очень просто, особенно если перевести в десятичную систему счисления:

$(71^5$*)(5^34$+^8-)

Эту команду можно ещё немного сократить. Попробуйте это сделать.

Изначально степлер не поддерживает записи отрицательных чисел, но если немного подумать, то их можно записать с помощью простого выражения. Достаточно просто вычесть из нуля число, чтобы получить его отрицательное значение. Вот, например, это выражение означает -3: 0^3-.

АХТУНГ!

Операцию 9^0/ , впрочем как и 6^0%, а также как 0^7-@ делать ни в коем случае нельзя. Если не знаете почему, советуем вам почитать школьный курс математики 3-го класса. В противном случае – я не виноват.

А теперь представим такую ситуацию: нам надо занять несколько подряд идущих переменных различными значениями. Для этого можно просто написать несколько команд присваивания, но можно поступить и более элегантно: если во время вычисления ОПЗ в стеке остаются значения, то эти значения копируются в следующие ячейки.

На практике это будет выглядеть вот так:

```
$(3)(7^8^25)
```

После выполнения этой команды третья ячейка будет равна семи, четвёртая – восьми, а пятая – двадцати пяти. Вот так-то всё просто.

2.4.3 Логика

У степлера есть также команды для побитовых логических операций. Но прежде чем их изучать, стоит, наверное, разобраться с тем, как в памяти степлера представляются числа. Каждое число в степлере является 16-ти битовым знаковым, то есть 15 битов отводятся под число, и один бит отводится под знак. Если этот бит сброшен (то есть равен нулю), то число положительное, а если же установлен (равен одному), то число отрицательное. Это всё показано на рисунке 01h.

Это всё надо знать для того, чтобы не быть удивлённым, когда в результате логической операции над двумя положительными числами получается отрицательное. То же самое может быть и при сложении или умножении чисел, когда результат превышает максимально возможный.

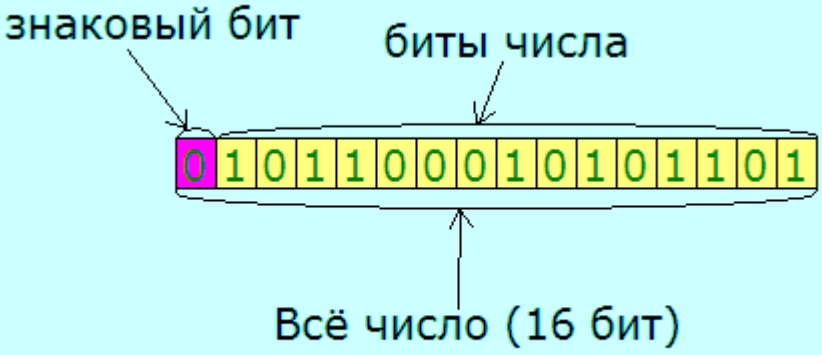


Рис. 01h Представление числа

А вот и сами логические операторы:

. – Логическое И над вершиной и подвершиной.

\ - Логическое ИЛИ.

: - Отрицание вершины.

- Исключающие или.

На эти операторы действуют те же правила, что и на арифметические операции.

Вот парочка примеров: ~b10001^~b100. – это будет двоичное число 10101 или десятичное 21. И ~hFFFF\ - это ноль. Так-то вот

2.4.4 Другое

Также есть ещё несколько достаточно интересных операторов, которых нельзя классифицыровать ни как логические, ни как арифметические.

! – Получает знак числа. Если на вершине стека положительное число, то он его заменяет на -1, если же положительное – на 1, а если же 0, то он его так и

оставляет. Этот оператор очень часто будет применятся при сравнении двух чисел, но об этом позже.

| - Отрицание на ноль. Если на вершине стека 0, то он заменяется на один, а если же не ноль, то число на вершине стека заменяется на 0. А эта команда будет тоже часто использоваться при сравнении и создании условий.

? – Случайное число от нуля до числа на вершине стека минус один. Например, если на вершине стека 100, то это число заменится на случайное от 0 до 99.

Чтобы получить случайное число от одного до числа на вершине стека надо сделать вот такую простую конструкцию: ?^1+ Там сгенерируется случайное число от нуля до числа на вершине стека, а потом прибавится один и получится то, что нужно.

А сейчас начнётся самый огонь: команда самомодификации. Она обозначается двойными кавычками (") и делает вот что: она берёт значение из стека, превращает его в оператор (по его ASCII-коду), и выполняет его. Таким образом, программа на степлере может сама себя модифицировать.

Например, вот небольшая программка, в которой используется этот оператор:

```
$(2)(2$^2$^1$")
```

Что делает эта программа? А вот что – то же самое, что и эта программа на паскале:

```
Program calc;
Uses crt;
Var x1,x2:integer;
C:char;
Begin
```

```
Readln(x1);
Readln(x2);
C:=readkey;
Case c of
'+':writeln(x1+x2);
'-':writeln(x1-x2);
'*':writeln(x1*x2);
'/':writeln(x1 div x2);
End;
Readln;
End.
```

А если честно, то это калькулятор, который просит ввести с клавиатуры два числа, а потом знак действия, которое над ним сделать, а потом он выполняет это действие и выводит результат на экран. И всё это всего в одну строчку. Так-то!

Ну в общем, это все операторы. Есть, правда, ещё парочка, но о них пока рано говорить, потом их обсудим, а пока этих операторов вполне достаточно, чтобы сделать вычисление любой сложности, хоть доказать что 2x2=5. Операторы – это сила!

2.5 Команды управления

В степлере, как и в любом другом нормальном языке, есть команды управления. Но в отличии от других языков программирования, в степлере нет условных блоков, но зато есть команды условного перехода, условного вызова процедуры и их безусловные аналоги.

Сейчас мы рассмотрим команды переходов. Для этого надо определиться, куда мы будем переходить. Мы будем переходить на метку. Метка – любое место в программе, которому мы можем дать имя, пометить.

Метки означаются вот так:

{имя метки}

Вот пример:

{метка}

Теперь, чтобы программа перешла на эту метку, нужна команда перехода, и вот она:

#<метка>

Если программа встретит такую конструкцию, она обязательно перейдёт на ту метку, которая там указана.

АЙ-АЙ-АЙ!

Нельзя делать переход на несуществующую метку. В противном случае будет то, что нарисовано на картинке или типа того. Если что – я не виноват.

Вот пример программы с этой командой:

Program

#<label>

[ыыыыыыыы]

{label}

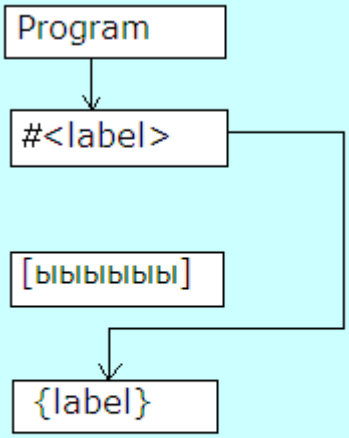


Рис 02h -блок схема программы.

Рассмотрим эту программу поподробнее. Вначале стоит команда Program, обозначающая начало программы, потом написана команда безусловного перехода на метку label. Программа переходит на эту метку, пропуская команду вывода текста. Блок-схема этой программы представлена на рисунке 02h.

Ещё одна интересная программка, в которой также есть немного математики:

```
Program
$(3)(0)
{next}
$(3)(3$^1+)
$(2)(3$)
#<next>
```

Эта программка немного сложнее, это заметно, но она уже похожа на типичные степлеровские программы.

Попытаемся с ней разобраться. Блок схема этой программы нарисована на рис. 03h.

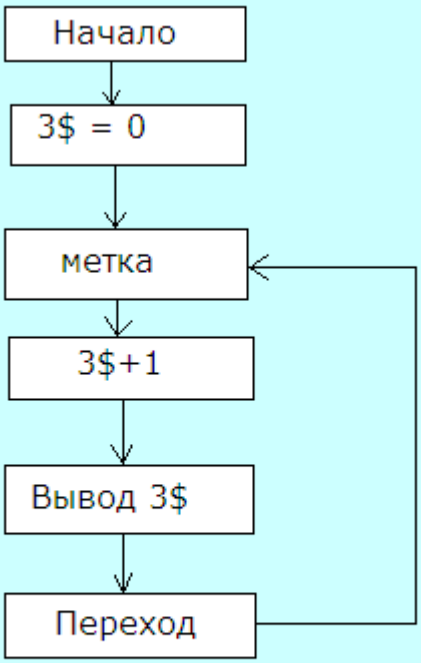


Рис. 03h

После команды Program находится команда присваивания, которая присваивает третьей переменной значение 0, то есть обнуляет её. Далее идёт метка next, а после неё ещё одна команда присваивания, которая присваивает третьей переменной её же значение плюс один, то есть увеличивает её значение на один, потом это значение выводится на экран, методом записи во вторую ячейку.

После того, как мы вывили число на экран, стоит безусловный переход, который отбрасывает программу к началу, где опять третья переменная увеличится на один и

выведется на экран, потом безусловный переход опять её отбросит к началу, и так много-много-много раз. В итоге на экран будет выводится последовательность чисел: 1, 2, 3, 4 … Эта последовательность будет идти бесконечно, то есть никогда не кончится, остановить программу можно только нажатием на резет или выключением интерпретатора, но есть одна загвоздка, поскольку степлер не может отобразить числа более 32768, программа будет работать вот так: 1, 2, 3, 4, …, 32766, 32767, 32768, -32767, -32766, …, -3, -2, -1, 0, 1, 2, 3, и так по кругу.

Такие вот конструкции, где одна часть кода повторяется несколько раз, называются циклами. Потом мы научимся делать различные типы циклов, но все они сделаны с использованием команд условного перехода, поэтому сейчас мы о них и поговорим.

Команды условного перехода это почти то же самое, что команды безусловного перехода, только с одним отличием: переход производится не всегда, а только при определённых условиях.

Посмотрим, как это пишется.

#(условие)<метка>

Как видим, отличий от команды безусловного перехода мало – появилось только поле «условие» между знаком решётки и полем метки. В это поле надо записать число или числовое ОПЗ выражение, переход по метке будет производиться только тогда, когда выражение в круглых скобках (условие) будет равно нулю.

Например, вот эта команда будет делать то же самое, что и команда безусловного перехода:

#(0)<метка>

Как мы знаем, переход на метку осуществляется только тогда, когда в поле «условие» стоит ноль, а поскольку там ноль, то переход производится всегда.

А теперь рассмотрим такую команду:

#(3$)<метка>

Что она делает? А вот что: переход производится только тогда, когда в третей ячейке ноль. Это и есть условный переход.

А теперь напишем простую программку для закрепления этих знаний. Она будет считать от определённого числа до нуля. Для начала лучше нарисовать блок-схему этой программы, так нам будет легче написать саму программу, да и вообще, прежде чем реализовать какой-то алгоритм, лучше нарисовать (или хотя бы представить в уме) его блок-схему. Вот блок схема на рис. 04h:

Будем составлять программу по этой блок-схеме.

Первое. Начало. Ну, естественно, это будет команда Program.

Program

Второе. Установить счётчик. Здесь надо решить, какую переменную мы будем считать за счётчик. Пускай это будет третья переменная, а считать мы будем, начиная с 10. Тогда команда будет выглядеть так:

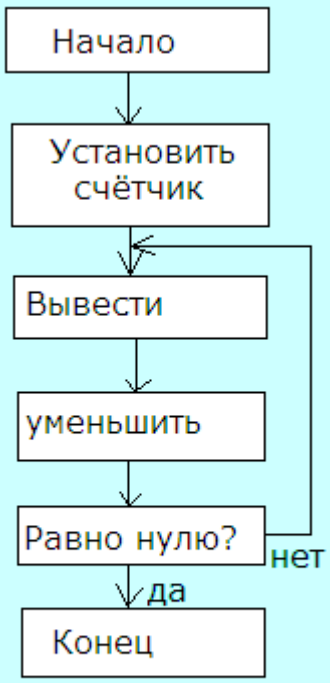


Рис. 05h

```
$(3)(10)
```

На блок-схеме мы видим, что после установки счётчика туда подходит стрелка. Это значит, что туда будет производиться переход, то есть, там будет метка. Назовём её labl

```
{labl}
```

Третье. Вывести счётчик. Тут, думаю, вопросов не будет.

```
$(2)(3$)
```

Четвёртое. Уменьшить. В этой ситуации надо поступать почти так же, как мы и поступали с предыдущим примером, только там мы увеличивали значение переменной на один, а тут нам надо уменьшать.

```
$(3)(3$^1-)
```

Пятое. Если не равно нулю, то перейти. Вроде тут всё просто, но есть один подвох: здесь надо перейти на метку, если НЕ равно нулю, а у нас есть только команда переход, если равно нулю. Что делать в этом случае? А всё просто, у нас есть оператор отрицания на ноль (|), и если его поставить после условия, то оно меняется на противоположное. Например, если без этого оператора переход проводится, если «РАВНО нулю», то если поставить этот оператор, будет переход «НЕ РАВНО нулю». Так-то вот. Поэтому команда будет выглядеть:

```
#(3$|)<labl>
```

Шестое. Конец. Ну, конец это конец.

Теперь можем написать всю программу полностью.

```
Program
$(3)(10)
```

```
{labl}
$(2)(3$)
$(3)(3$^1-)
#(3$|)<labl>
```

Как видите, программа достаточно короткая, а на её описание мы потратили аж две страницы. Ну что поделать, только так можно научиться писать программы.

Попробуйте изменить предыдущую программу так, чтобы она требовала ввод числа, а потом считала от этого числа до одного.

Пока всё ☺

2.6 Процедуры

Для того, чтобы сократить размер программы, не записывая один и тот же участок кода в нескольких местах программы, нужно использовать процедуры. Процедуры – это специальные участки кода, которые можно вызывать на выполнение из любого места программы. Так сказать, это небольшие программки, которые находятся внутри основной программы. Схематически принцип вызова подпрограмм нарисован на рис. 06h

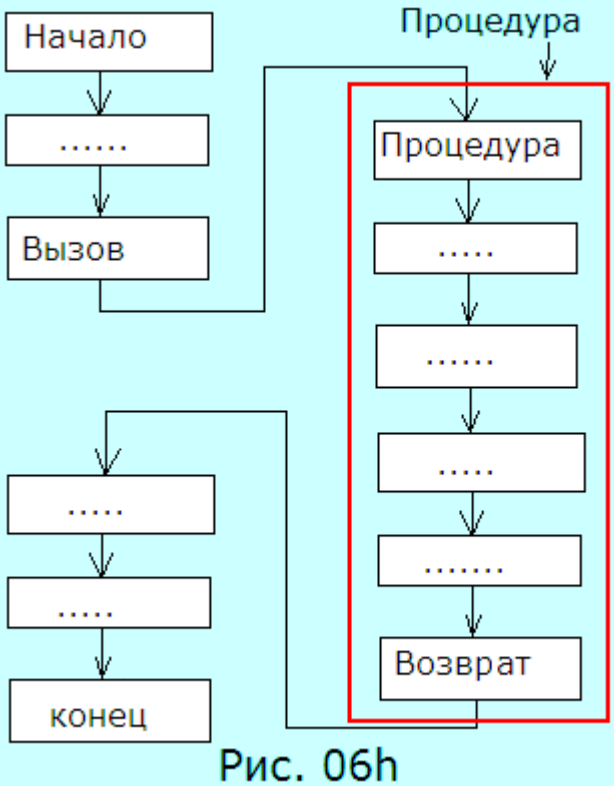


Рис. 06h

Что мы там видим?

Мы там видим, что после команды вызова процедуры, поток выполнения основной программы прерывается, и управление

передаётся процедуре. Далее выполняются команды, которые находятся внутри процедуры. Когда внутри процедуры интерпретатор встречает команду возврата, то выполнение процедуры заканчивается, и выполнение передаётся в то место, в котором была вызвана процедура.

Именно так теоритически производится запуск процедуры. Теперь разберёмся с тем, как всё это реализовать на степлере.

Процедуры описываются вот так:

```
={имя подпрограммы}=
команда
команда
…
команда
**
```

Первая команда в процедуре имеет следующий формат: вначале пишется знак равно, потом открывающаяся фигурная скобка и потом, без пробелов, имя процедуры. Надо помнить, что названия процедур не должны совпадать в названиями меток. Далее, после имени процедуры пишется закрывающаяся фигурная скобка и знак равно. Вот так описывается заголовок процедуры.

После заголовка пишутся обычные команды, которые описывают какой-либо алгоритм или действие. После того, как процедура закончилась, надо написать команду завершения подпрограммы или возврата в основную программу. Эта команда обозначается двумя звёздочками: **.

А в какой части программы располагаются процедуры? Серьёзный вопрос. До этого момента мы знали вот такую структуру программы на степлере:

Program

Программа

Теперь мы должны внести некоторые изменения, а именно добавить блок, в котором располагаются все процедуры. Этот блок находится до команды Program, то есть до начала самой программы. Соответственно, теперь структура программы будет выглядеть так:

Подпрограммы

Program

Программа

А как же вызывать эти подпрограммы? Их вызывать очень просто, и, возможно, вы немного удивитесь, вы уже почти умеете это делать. Программа вызова процедуры очень похожа на команду перехода на метку. Вот команда безусловного вызова:

*<имя процедуры>

А вот команда условного:

*(условие)<имя процедуры>

Как видите, отличий действительно мало, заменён только первый символ команды (# на *).

Чтобы разобраться во всём этом немного глубже, напишем наипростейший пример. Этот пример будет содержать процедуру, которая выводит на экран надпись, а основная программа будет вызывать эту процедуру. Напишем вначале эту процедуру, пусть она называется

outstr, и она будет выводить на экран надпись «Винни-пух лучший!».

```
={outstr}=
[Винни-пух лучший!]
**
```

Поскольку это единственная процедура в этой программе, то можно писать команду начала программы.

```
Program
```

Теперь нам нужна команда безусловного вызова этой подпрограммы. Это сделать просто:

```
*<outstr>
```

Теперь можно уже написать полностью то, что у нас получилось.

```
={outstr}=
[Винни-пух лучший!]
**
Program
*<outstr>
```

Про условный вызов процедур говорить подробно не будем, так как он ничем не отличается от условного перехода на метку, это мы уже изучали.

Надо также отметить о том, что процедуры могут вызываться не только из основной программы, но и из других подпрограмм. Вот небольшой пример, иллюстрирующий это:

```
={aaa}=
[output]
```

```
**
={bcd}=
*<aaa>
**
Program
*<bcd>
```

В этой программе есть две процедуры: aaa и bcd. При запуске программы производится вызов подпрограммы bcd, а подпрограмма bcd вызывает подпрограмму aaa, которая, в свою очередь выводит на экран надпись. Конечно, эта программа очень непрактична, её можно достаточно сильно сократить, и получится программа размером всего в две строчки. Кстати, попробуйте это сделать.

А теперь давайте поговорим о чисто техническом аспекте вызова подпрограмм: как это происходит с точки зрения интерпретатора? Этот материал не обязателен к прочтению, можете его пропустить, если не интересно, хотя если вы его прочитаете, вы будете достаточно хорошо понимать, как происходит вызов подпрограмм.

Во время изучения записи числа мы сказали, что у степлера четыре (или три) стека. Один мы уже изучили, а про остальные сказали, что изучим потом. Так вот, для одного стека уже пришло время. Это стек возвратов. Он используется для вызова подпрограмм, и сейчас мы разберёмся, как он это делает.

Стек возврата – это стек, в котором запоминается адрес, куда надо возвратиться после завершения процедуры. Сейчас это будет разъяснено подробнее. Рассмотрим алгоритм вызова процедуры:

1.Поместить в стек возврата текущее положение программы.

2.Произвести безусловный переход на начало процедуры.

Как видим, всё предельно просто. Кстати, точно также устроен вызов подпрограммы у всех современных процессоров. Теперь изучим алгоритм выхода из процедуры. Тут можно догадаться и самому, но мы лучше объясним. Для возврата из процедуры нам надо знать адрес того места, с которого мы её вызывали, чтобы туда возвратиться. А этот адрес как раз и находится на вершине стека возвратов. Оттуда мы и берём адрес, и возвращаемся туда. Вот так.

Таким образом, при вызове процедуры в этот стек записывается текущее положение программы, а при выходе из неё эти значения удаляются.

Попробуйте написать программку, которая выводит какую-нибудь надпись, если с клавиатуры ввели ноль, а если же ввели не ноль, то выводит другую надпись. В программе используйте процедуры.

2.6.1 Передача параметров

До этого мы делали простые процедуры, которые выводили на экран простую строку, но если нам нужна процедура, которая будет выполнять какие-то вычисления, то, соответственно, напрашивается вопрос, как передавать процедуре входные параметры и получать от неё результат. Конечно, это можно сделать с помощью специальных переменных, например, процедура берёт свои параметры из пятой, шестой, седьмой переменной, и возвращает результат в девятую переменную. Это очень

неудобно, так как используется много лишних переменных, которые могли бы использоваться для других целей. Для решения этой проблемы был сделан ещё один (то есть третий) стек – стек параметров.

Стек параметров – это специальный стек, который создан специально для передачи параметров в процедуры и для получения результата вычислений процедуры. Теоретически это выглядит так: перед вызовом процедуры в стек заносятся параметры этой процедуры. Потом, когда процедура запустилась, она из стека забирает эти значения и выполняет над ними действия. После завершения вычислений процедура заносит в стек результат своих действий, а основная программа оттуда забирает и использует по своему усмотрению. Такая система очень удобна, так как она не делает никаких ограничений, что нельзя пользоваться какой-то переменной, потому что она нужна только для передачи параметров для такой-то функции, а эти переменные тоже нельзя использовать, так как они для другой функции ,и так далее. Стек параметров автоматически решает эти проблемы. Научимся им пользоваться на практике.

Есть несколько команд записи данных в стек параметров. Сейчас мы их рассмотрим.

Первый, и самый распространённый метод передачи, это с помощью команды вызова подпрограммы. Когда мы изучали команды условного и безусловного вызова, мы умолчали об ещё одной способности этой команды – передачи данных в стек параметров. Ну, в общем это было правильно, мы тогда не знали об этом, значит можно было и не рассказывать.

Для передачи данных в команде перехода есть ещё одно поле – поле для записи данных. Это поле располагается после поля условия и заключается в квадратных скобках. Команда перехода со всеми возможными полями называется расширенной командой вызова процедуры и выглядит так:

*(условие)[параметры]<имя процедуры>

В поле «параметры» находятся числа, передающиеся в стек, разделяемые запятыми. Каждое число может быть не просто числом, а выражением ОПЗ.

Вот парочка примеров:

*[7,8]<proc>

*[4$]<proc>

*[2^8$+,989]<proc>

Думаю, не стоит объяснять, что делают эти команды, уточню лишь одно: данные заносятся в стек слева направо. То есть, после выполнения первой команды из списка выше на вершине стека будет 8, а на подвершине будет 7.

Если же нам надо просто передать в стек параметров какое-то число, не вызывая процедуры, для этого есть специальная команда, и вот её синтаксис:

P(числа)

Вначале идёт буква p (английская п), а потом в скобках находятся числа, передаваемые в стек. Метод записи чисел в этой команде абсолютно такой же, как и в расширенной команде вызова процедуры. Для примера приведём две идентичные конструкции. Вот:

*[9,22]<proc>

И вот:

```
P(9,22)
*<proc>
```

Эти две части программы делают одно и то же: записывают в стек два числа: 9 и 22, а потом вызывают команду proc, просто делают это разными методами.

А теперь давайте изучим, как получать данные из этого стека. Действительно, мы достаточно долго рассматриваем методы записи в него, а методы чтения ещё даже не начинали. Восполним этот пробел.

Для чтения из стека параметров есть специальный оператор g. Он берёт данные из стека параметров и заносит их в стек ОПЗ. Допустим, нам надо прочитать данные из стека параметров и сохранить их в третьей переменной. Команда будет выглядеть так:

```
$(3)(g)
```

Всё очень просто.

А вот теперь давайте закрепим полученные знания на примере. Пусть это будет процедура, которая берёт два параметра и возвращает их сумму. Назовём её summ.

```
={summ}=
```

Далее нужно немного подумать. Можно сделать так: взять первый параметр из стека, положить его в переменную, потом взять другой параметр. И положить его в другую переменную, а потом сложить и опять положить в стек параметров. В результате получаем:

```
$(3)(g)
$(4)(g)
$(5)(3$^4$+)
```

```
P(5$)
```

Такой код нас явно не устраивает, что-то надо менять, и менять тут есть что. Например, мы видим, что значение из стека параметров копируется сначала в третью переменную, третья переменная является одним слагаемым. Так не легче ли сразу значения стека использовать в качестве слагаемых? Да, это намного легче, и если это сделать, то команды вычисления принимают такой вид:

```
$(5)(g^g+)
P(5$)
```

Здесь есть тоже что сократить. Мы видим, что сумма значений перемещается сначала в пятую переменную, а потом, с помощью команды p записывается в стек. Намного более практичнее будет сразу записывать в стек сумму значений. Тогда, процедура будет выглядеть так:

```
P(g^g+)
```

Как видите, мы сократили размер программы с четырёх строчек до одной. Такой процесс называется оптимизацией. И мы будем изучать его на страницах этой книги. (Ну ясно, что на страницах, не под ними же ☺)

Далее надо написать команду выхода из процедуры:

```
**
```

Итого, у нас получилась процедура, складывающая два числа. Вот она полностью:

```
={summ}=
P(g^g+)
**
```

А теперь давайте напишем программку, которая просит ввести с клавиатуры два числа, после ввода суммирует их с помощью этой процедуры, а потом выводит на экран сумму.

Вначале нужна команда Program.

```
Program
```

Потом, надо вывести сообщение, чтобы вы ввели два числа

```
[Введите два числа]
```

А теперь можно поступить двумя способами: можно первое число ввести в первую переменную, а второе число в другую, и эти переменные передать в стек параметров, а можно поступить другим, более рациональным способом, вот таким:

```
*[2$,2$]<summ>
```

Что делает этот метод? А делает он вот что: передаёт в стек значения двух вторых переменных, а что происходит при попытке чтения из второй переменной? Правильно, программа останавливается, и производится ввод числа с клавиатуры. То есть, когда в стек пытается передаться значение второй переменной, программа останавливается и ждёт ввода с клавиатуры. Когда число ввели, в стек помещается введённое число, и, по идее, в стек должно передаться следующие число, но это опять вторая переменная, то есть, происходит то же самое – ввод значения с клавиатуры.

Далее, когда сложение уже выполнено, надо вывести полученную сумму на экран, которую берём из стека возврата.

```
[Сумма= ]
```

```
$(2)(g)
```

Вот и всё! Получилась всё очень удобно и красиво. Вот вся программа:

```
={summ}=
P(g^g+)
**
Program
[Введите два числа]
*[2$,2$]<summ>
[Сумма=]
$(2)(g)
```

AAAAAAAAAa!

За стеком надо следить очень хорошо. Если вы вдруг нечаянно поместите лишнее число в стек или заберёте оттуда, если не клали – будет очень плохо, особенно при рекурсии. Если что – я не виноват.

Попробуйте изменить эту программу так, чтобы она не складывала два числа, а вычисляла сумму квадратов трёх чисел

2.6.2 Локальные переменные

Представим такую ситуацию: для решения некоторой задачи процедуре требуются переменные, в которых будут находится данные, которые будут использоваться только этой процедурой и нигде более. Что это может быть? Например, это может быть счётчик цикла, который выполняет какую-нибудь операцию, или ещё что-нибудь, где требуется место для хранения временных данных.

Для этого можно просто использовать обычные переменные, но нужно помнить о том, что эти переменные не должны использовать ни основная программа, ни другие подпрограммы. Вот почему: допустим, одна процедура использует какую-то переменную для хранения каких-то локальных значений. Она записала в эту переменную данные и вызвала другую процедуру, которая в качестве локальной переменной использует эту же переменную. Соответственно, эта процедура затрёт предыдущее значение, выполнит действия и возвратится в предыдущую процедуру, которая, в свою очередь, не сможет продолжить вычисления, потому что локальные переменные затёрты. Это называется конфликт памяти, и, чтобы его избежать, были придуманы локальные переменные.

Что же такое локальные переменные? Это переменные, значение которых может меняться во время работы процедуры, но с точки зрения основной программы значение этих переменных меняться не будет. Теперь рассмотрим, как это записывать. А вот так:

```
?[переменные]
```

Вначале пишется знак вопроса, а потом, в квадратных скобках, список переменных, которые будут локальными. Вот пример:

```
?[3,4]
```

Эта команда переменные 3 и 4 сделает локальными, то есть их значения можно будет менять во время работы процедуры.

Команда этого типа называется командой сохранения локальных переменных, и пишется она вначале процедуры, сразу после заголовка:

```
={proc}=
?[переменные]
Процедура
**
```

Напишем простую программу-пример. Пусть в ней будет процедура, которая изменяет значение какой-то переменной, которая является локальной. А основная программа после выполнения этой подпрограммы проверит значение этой переменной.

Вот эта вся программа:

```
={same}= ;Процедура
?[3] ;Объявляем третью переменную – локальной
$(3)(49) ;Записываем в третью переменную число 49
** ;Возврат
Program ;Начало программы
$(3)(77) ;Записываем в третью переменную число 77
*<same> ;Вызываем процедуру
$(2)(3$) ;Выводим на экран значение третьей ячейки.
```

Эта программа выведет на экран число 77, хотя мы видим, что внутри процедуры same этой переменной присваивается значение 49, но на работе основной программы это не отражается, потому что мы объявляем эту переменную локальной. Можете ради интереса убрать команду объявления локальной переменной, и тогда на экран выведется не 77, а 49, поскольку в процедуре same будет изменяться значение не локальной переменной, а глобальной.

А теперь разберёмся, как эта команда работает и почему она называется «команда сохранения значений локальных переменных». Для работы этой команды есть ещё один специальный стек: стек сохранения локальных переменных, и работает он вот так: во время выполнения команды сохранения локальных переменных интерпретатор берёт из списка локальную переменную, и в этот стек сохраняет её номер, а потом её значение. Так происходит для всех переменных, которые мы обозначаем локальными. Получается, мы сохраняем значения локальных переменных, какими они были до вызова процедуры. Далее, мы оперируем с этими переменными, изменяя их значения. Когда же интерпретатору попадается команда возврата из процедуры, он читает из стека переменной, значение которой мы сохраняли, а потом уже её значение, и так все переменные, которые мы сохраняли вначале процедуры.

Надеюсь, теперь вы разобрались, как происходит объявление локальных переменных.

А теперь давайте посмотрим на один вопрос: Сколько у нас уже стеков?

1.Стек ОПЗ

2.Стек возвратов

3.Стек передачи параметров

4.Стек сохранения локальных переменных

Как мы видим, у нас четыре стека, так почему же мы говорили, что у нас или четыре или три стека? А вот почему: многие интерпретаторы объединяют стек локальных переменных и стек возвратов. Зачем? Для экономии памяти. Поскольку тогда для интерпретации

нужно создавать всего три стека против четырёх. Как же две информации совершенно разного типа могут мирно сосуществовать вместе? А вот так: когда интерпретатор встречает команду вызова процедуры, он помещает в стек адрес возврата и переходит по адресу процедуры. Если он же находит в процедуре команду сохранения локальных переменных, то копирует в стек переменные и специальный маркер, чтобы интерпретатор знал, что локальные переменные объявлялись. Таким образом, стек имеет такую структуру: на вершине стека лежит маркер, потом переменные, а потом адрес возврата (рис. 07h)

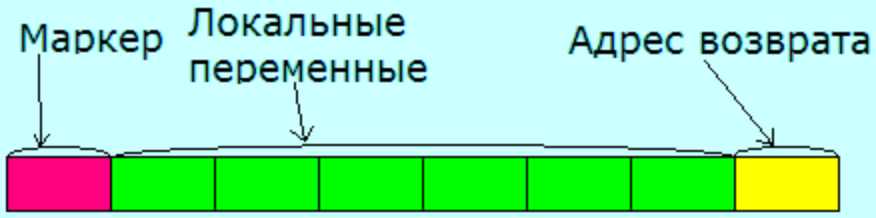


Рис. 07h Блок возврата

Что же происходит, когда интерпретатор встречает команду возврата? Прежде всего интерпретатор смотрит, не является ли вершина стека маркером? Если она является им, то вначале восстанавливаются локальные переменные, а потом производится переход по адресу возврата. Если же маркера нет, то происходит просто переход. Вот так!

2.7 Специальные функции

Тот степлер, который мы изучали до этого, был очень примитивен с точки зрения возможностей ввода/вывода. Он поддерживал только консольный текстовой режим, что нам очень и очень не нравится. Но, к нашему счастью, в степлере есть специальные функции, которые намного расширяют его возможности как практического языка программирования. С помощью этих функций можно

получить доступ к файлам, графике, прерываниям и многому другому.

Сейчас мы научимся их использовать. Все эти команды имеют один формат:

~(номер функции)(параметры)

Вначале идёт загогулина, потом, в скобках, номер функции, а, потом, тоже в скобках, параметры этой функции. Вот пример:

~(3)(100,45,96,89,7)

Всего есть 20 таких функций, пронумерованы они по порядку, от 1 до 20. Сейчас мы с ними познакомимся.

2.7.1 Графика и звук

Для работы с графикой в степлере есть несколько специальных функций. Но перед тем, как с ними разбираться, по-моему, стоит уяснить, какой графический режим мы будем использовать. Мы будем использовать режим 640 на 480 точек и 256 цветов. Этого вполне достаточно для рисования относительно сложных объектов. Структура экрана в графическом режиме представлена на рисунке 08h. У каждой точки есть свои координаты, и определяются они двумя числами, которые обозначаются буквами X и Y. X – это расстояние от левого края экрана, а Y – это расстояние от верхнего края.

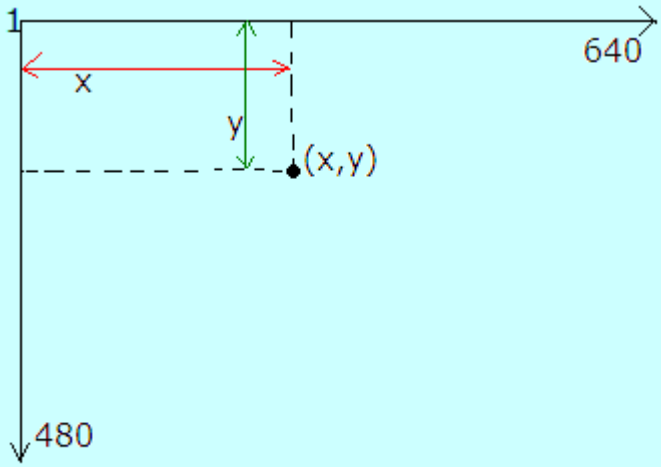


Рис.08h Структкра экрана

Теперь рассмотрим уже сами команды. Первая и

самая главная команда – установка графического режима. Она имеет следующий формат:

~(1)(режим)

Здесь режим – это номер графического режима. Есть два режима – это графический (1) и текстовой (0), то есть, чтобы включить графический режим, надо написать:

~(1)(1)

А чтобы его выключить:

~(1)(0)

Цвет фона при включении экрана будет чёрным.

Надо учесть, что в графическом режиме не работают обычные команды вывода информации, такие как запись в первую и вторую ячейки, а также вывод строки, для этого будут другие соответствующие команды в графическом режиме. Теперь, когда мы уже научились устанавливать графический режим, пора научиться рисовать. Первой фигурой, которую мы изучим, будет точка. Обычная точка, размером в один пиксель. Она выводится такой командой:

~(2)(X,Y,цвет)

Как мы видим, у команды есть три параметра: первые два нам уже знакомы – это положение точки(см. Рис. 08h), а вот с третьим параметром мы ещё не встречались. Вообще-то, значение этого параметра можно и не объяснять, это и так понятно – он обозначает цвет пикселя. Вот пример команды, которая рисует зелёную точку в центре экрана:

~(2)(320,240,10)

320 и 240 – это положение точки, а 10 – это номер зелёного цвета. Номера всех цветов даны в приложении книги.

Теперь надо разобраться ещё с одной простой командой – рисование линии. Линия описывается в степлере, как две точки: её начало и конец. Поэтому структура команды:

~(3)(Хнач,Унач,Хкон,Укон,цвет)

Думаю, всё понятно. Хнач и Унач – это координаты начала линии, а Укон и Хкон – её конца. Цвет это цвет. Вот пример горизонтальной линии, пересекающей экран:

~(3)(0,240,640,240,10)

Ещё одна важная фигура – круг. В степлере её тоже можно строить. Если вы разобрались с предыдущими командами, а также если у вас есть хоть какие-то знания в геометрии, то вы это тоже поймёте. Круг строится по координатам центра, радиуса и, соответственно, цвету. Вот структура:

~(4)(X,Y,радиуис,цвет)

Пример, думаю, приводить не стоит.

Вот и все фигуры, которые можно нарисовать в степлере стандартными методами. Да, вы скажите, что есть ещё много фигур, например, треугольник или квадрат, но их можно нарисовать с помощью существующих объектов или сделать процедуру. Вот, например, процедура, рисующая прямоугольник по двум его точкам (Рис.09h)

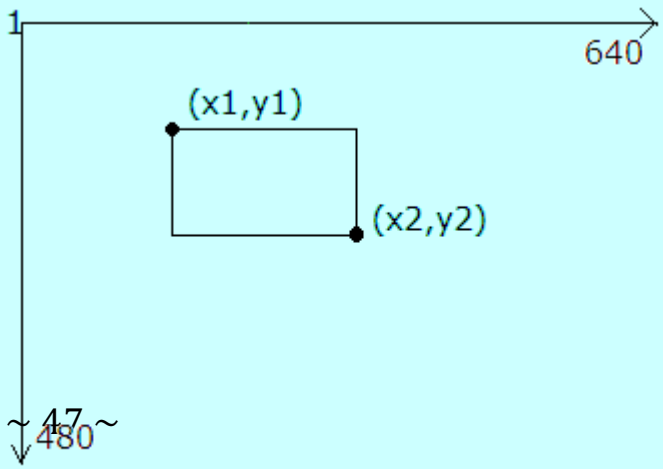


Рис.09h Прямоугольник

```
={drawrectangle}=
?[3,4,5,6,7]
$(3)(g^g^g^g^g)
~(3)(5$,6$,7$,5$,3$)
~(3)(7$,5$,7$,6$,3$)
~(3)(7$,6$,5$,6$,3$)
~(3)(5$,6$,5$,4$,3$)
**
```

Эту процедуру нужно вызывать вот так:

```
*[X1,X2,Y1,Y2,цвет]<rectangle>
```

Вот такую процедуру можно сделать для всех других геометрических объектов, мы не будем этого делать, а просто научим вас делать процедуры. Но об этом потом.

Иногда возникает ситуация, что надо не только нарисовать что-то на экране, а наоборот, узать цвет какого-либо пикселя. Для этого есть команда получения цвета пикселя. Она имеет следующий формат:

```
~(5)(X,Y,переменная)
```

X и Y – это координаты пикселя, цвет которого мы должны узнать, а «переменная», это номер переменной, в которую мы запишем найденное значение. Вот простой пример:

```
~(2)(50,76,14)
~(5)(50,76,7)
```

Что мы тут видим? Первая команда рисует точку 14-ого цвета (жёлтый) на экране. Вторая команда читает цвет этой точки, и результат помещает в седьмую переменную. Естественно, в седьмой переменной будет число 14.

Теперь давайте поговорим об одной такой важной команде как заливка. Она позволяет намного сократить количество операций рисования, особенно, когда надо рисовать большие однородные поверхности. Команда имеет такой формат:

~(17)(X,Y,цвет заливки, цвет границы)

Про X и Y умолчим, надеюсь, что вы уже знаете, что это такое, цвет заливки, тоже, думаю, понятно, а, вот про цвет границы надо поговорить поподробнее. Дело в том, что заливка в степлере немного отличается от той, что в Paint'е. Границей здесь являются не все точки другого цвета, а только точки определённого цвета. Если говорить простым языком, то область, которую мы будем закрашивать, должна быть обрамлена линией одного цвета, и этот цвет мы должны указать.

Примерчик: нарисуем белый круг с ободком зелёного цвета. Вначале надо нарисовать границу, или ободок. Зелёный цвет имеет номер 10. А потом залить эту окружность белым цветом (15-ым), указав при этом цвет границы зелёный. Получится вот так:

~(4)(100,100,50,10)

~(17)(100,100,15,10)

Думаю, всё понятно.

А теперь давайте поговорим об ещё одном типе вывода информации в графическом режиме – выводе символьной информации. Для этого есть три команды:

Вывод символа:

~(20)(цвет, размер, X, Y, код символа)

Вывод числа:

~(19)(цвет, размер, X, Y, число)

Вывод строки:

~(18)(цвет, размер, X, Y, смещение к началу строки)

С первой и второй командой, думаю, всё понятно, а вот с третьей командой надо разобраться немного посерьёзнее.

В ней используется параметр «смещение к началу строки». Что это такое? А это число, которое указывает на переменную, которая является первым символом строки, а сама строка должна оканчиваться нулём. Приведём пример. Пускай эта строка выводит слово СТЕПЛЕР. Для этого нам нужно затолкать в память это слово, и к концу приписать ноль. Это можно сделать, присваивая значения ячейкам побуквенно, но можно поступить и более изящным способом:

$(9)('СТЕПЛЕР'^0)

В этой команде используется принцип, что если в стеке остаются значения, то они копируются в следующие ячейки. Здесь копируется слово СТЕПЛЕР, и ноль на конце, обозначающий конец строки. После выполнения этой команды, память будет выглядеть как на рис. 0Ah.

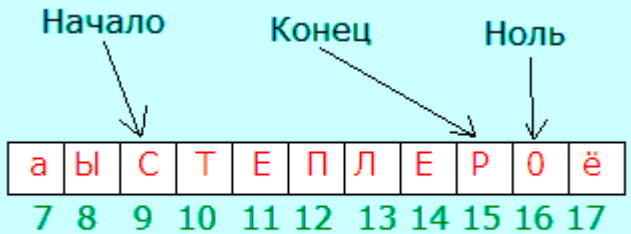


Рис.0Ah Строки

Теперь, когда мы записали строку в память, нам надо её вывести на экран. Воспользуемся такой командой:

~(18)(14,3,100,100,9)

Вот так вот выводятся строки, а сейчас мы рассмотрим ещё две команды, которые ничего не рисуют, а просто управляют графическим процессом.

Первая команда – очищение экрана. Она имеет такой формат:

~(16)(0)

Никаких параметров нет, она просто очищает экран, делая его чёрным.

Есть ещё одна, более сложная команда, которая просто необходима при рисовании сложных, многоцветных объектов. Эта команда переназначение цвета. Что она делает? А вот что – переназначает какой-нибудь цвет палитры. Например, 10-ый цвет у нас зелёный, а нам надо, чтобы он был красный или синий. Это можно сделать с помощью этой команды. Для чего это нужно? Например, чтобы изменить палитру под свои нужны, сделать много оттенков серого, или создать совершенно другую палитру, что нужно, например, для программы просмотра BMP-файлов.

Теперь давайте рассмотрим эту команду. Она имеет такую структуру:

~(14)(номер цвета, красный, зелёный, синий)

Как мы видим, вначале идёт номер цвета, который мы хотим переназначить, потом его красная составляющая, потом зелёная и синяя. Каждая составляющая может принимать значения от 0 до 63. Вот пример команды, которая переназначает 14-ый цвет (жёлтый) на красный:

~(14)(14,63,0,0)

После выполнения этой команды все объекты, которые вы будете рисовать жёлтым цветом, будут рисоваться красным, и даже все уже нарисованные этим цветом рисунки станут красными. Это свойство можно использовать для быстрой анимации, нарисовать рисунок каким-нибудь 156 цветом, а потом менять значение этого цвета, переназначать его много раз, и будет казаться, что рисунок меняет цвет. Поверьте, этот способ намного быстрее, чем заново перерисовывать его другим цветом.

Ну вот и всё, с графикой закончили, теперь надо разобраться со звуком.

Для работы со звуком есть одна команда:

~(6)(частота)

Она начинает играть звук с заданной частотой и перестаёт его играть только тогда, когда встретит команду выключения звука:

~(6)(0)

Ещё одна команда, которая часто используется вместе с этим командами – команда паузы. Она выглядит так:

~(15)(время)

Эта команда приостанавливает выполнение программы на определённое время, которое указывается в сотых долях секунды.

Напишите программу, рисующую домик, в котором выключен свет, но после нажатия на энтер в доме зажигается свет, а после следующего нажатия открывается дверь.

2.7.2 Работа с файлами

Команды работы с файлами не блещут таким разнообразием, как команды работы с графикой, но всё равно, есть всё необходимое.

Вначале надо рассмотреть самую главную команду, которая открывает доступ к файлу. Она имеет три подфункции:

```
~(10)(смещение, номер, 0)
```

Открыть файл для чтения. Имя файла должно находиться в памяти и начинаться с ячейки «смещение». Параметр «номер» означает номер файла. Мы должны знать номер каждого файла, который мы открывали, потому что при чтении/записи в файл мы должны его указывать. Надо также учесть, что открываемый файл должен существовать, при попытке открыть несуществующий файл будет выдана ошибка.

Вторая подфункция – открыть файл для записи.

```
~(10)(смещение, номер, 1)
```

Тут всё похоже, акцентироваться особо не будем, скажем лишь, что если файл существует, то он очистится, и на его месте будет создан пустой, а если он не существует, то файл будет создан.

Ну и, наконец, третья подфункция – перемещение в открытом файле:

```
~(10)(позиция, номер, 2)
```

Тут мы видим, что параметр «Смещение» изменён на «позиция». Это не случайно, ведь у этой подфункции немного другое назначение: не открывать файл, а перемещаться внутри уже открытого.

С открытием файла мы уже разобрались, теперь надо разобраться с чтением/записью в файл. И чтение и запись происходят в побайтовом режиме, то есть, из файла за один раз можно прочитать только один байт, и записать можно тоже только один байт.

Рассмотрим команду записи в файл. Она имеет следующую структуру:

~(11)(номер файла, байт)

Вначале идёт номер файла, в который мы будем записывать байт, а потом уже сам байт, который мы должны туда записать. Для нормальной работы этой команды файл должен быть открыт для записи.

Теперь давайте рассмотрим команду чтения из файла. Она похожа на команду записи и имеет такую структуру:

~(12)(номер файла, ячейка-приёмник, признак конца)

Разберёмся поподробнее: что такое «номер файла», думаю, понятно, «ячейка-приёмник» - это номер переменной, в которую поместится прочитанный байт, а в параметр «признак конца» пишется номер переменной, в которую поместится 0, если прочитан последний байт файла, или 1, если не последний. Для работы этой команды файл должен быть открыт для чтения.

После каждого прочтения байта позиция чтения увеличивается на один, то есть мы прочитали первый байт, потом, при следующем прочтении, будет второй, третий и так далее. А если нам надо прочитать какой-то байт из середины файла, не прочитывая при этом все предыдущие байты? Для этого у нас есть команда, которую мы уже изучали – команда перемещения по файлу.(Подфункция 2 функции 10)

И ещё одна команда, закрывающая файл:

~(13)(номер файла)

Её надо писать после каждой работы с файлами, потому что если не закрыть файл после записи, то файл или будет пустым, или вообще открываться не будет.

Напишем простой пример: пусть это будет программа, которая выводит файл на экран. Блок-схема программы на рисунке 0Bh. Напишем по ней программу.

Первое. Начало. Ну это команда Program.

Program.

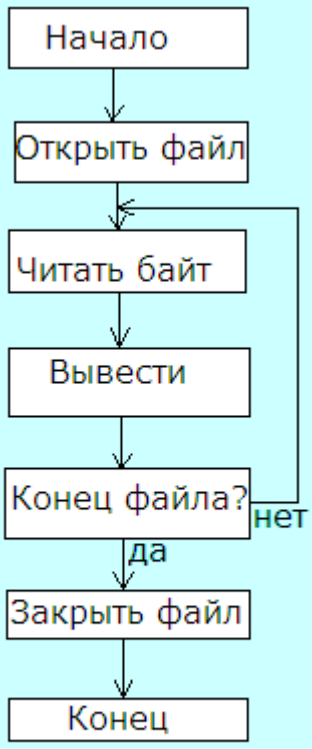


Рис.0Bh

Второе. Открыть файл. Для этого надо две команды: первая задаёт имя файла, а вторая открывает его для чтения. Пусть это будет файл text.txt, и имя будет начинаться с пятой переменной.

$(5)('file.txt'^0)

~(10)(5,1,0)

Далее идёт метка. Назовём её readbyte

Третье и четвёртое. Прочитать и вывести. Эти два пункта можно было бы сделать отдельно, но лучше их объединить. Это можно сделать так: в качестве приёмника байта указать первую переменную, и все прочитанные байты будут вылазить на экран. В качестве переменной-детектора конца файла будем использовать четвёртую переменную.

~(12)(1,1,4)

Пятое. Если не конец файла, то перейти к метке. Четвёртая переменная – у нас детектор конца файла, и

если бы нам надо было перейти, если файл кончился, то в условие мы могли бы просто написать 4$, но нам нужен переход, если файл не кончился, поэтому после условия надо поставить |, и тогда всё будет верно.

```
#(4$|)<readbyte>
```

Шестое и седьмое. Закрыть файл и конец. Закрываем файл обычной командой, а конец – это делу венец.

```
~(13)(1)
```

Теперь можно написать всю программу вместе:

```
Program
$(5)('file.txt'^0)
~(10)(5,1,0)
{readbyte}
~(12)(1,1,4)
#(4$|)<readbyte>
~(13)(1)
```

Вот и всё. При запуске этой программы на экран будет выведен текст файла file.txt, только чтобы он правильно вывелся, он должен либо не содержать русских букв, либо должен быть написан в досовской кодировке, иначе будут выведены кракозябы.

АХТУНГ!

С помощью этих команд ещё как легко затереть какой-то важный файл, и это может привести от лёгкого недоумения до полной потери всей информации. Если что – я не виноват.

2.7.3 Работа с системой

А вот, наверное, самый интересный раздел. Здесь мы научимся управлять системой напрямую, а именно: писать в порты и читать, вызывать прерывания и делать прямые пересылки в памяти. Для чего это всё нужно? – Как для чего? – Для создания супер-пупер-мега крутых программ, чтобы использовать все возможности системы, писать вирусы, и вообще, делать всё, что угодно, ведь это прямой доступ к системе.

Спустимся с небес на землю и будем изучать уже сами команды. И первая из них- команда записи в порт ввода/вывода. Она имеет следующий формат:

```
~(7)(номер порта, значение)
```

Эта команда выполняет запись в порт ввода/вывода. Она пересылает значение «значение» в порт «номер порта». Например, вот эта команда пересылает на параллельный порт ASCII-код буквы E. Это значит, если бы к порту был подключён принтер, то он бы напечатал эту букву.

```
~(7)(~H378,'E')
```

Поскольку мы уже научились писать в порты, то, думаю, нам уже следует научиться читать из них. И сейчас мы это сделаем. Команда чтения из порта имеет следующий формат:

```
~(8)(номер порта, номер переменной-приёмника)
```

Вначале мы видим «номер порта» - это порт, из которого мы будем читать. Потом мы видим «номер переменной-приёмника», в эту переменную поместится прочитанное значение. Вот, например, команда, которая читает значение из порта клавиатуры и заносит его 5-ую переменную.

~(8)(~H60,5)

Нет, это не аналог команды $(5)(1$), эта команда намного круче и позволяет сделать намного больше интересных вещей. Например, позволяет отлавливать клавиши типа Alt, Ctrl и Shift. Но об этом потом.

А теперь про ещё одну очень важную команду, команду вызова прерывания. Она имеет немного более сложный синтаксис:

~(9)(номер прерывания, смещение к регистрам)

Про «номер прерывания», думаю, понятно, а вот со «смещением к регистрам» нужно разобраться. Смещение к регистрам показывает переменную, с которой начинают располагаться значения регистров, и они будут располагаться в такой последовательности: ah, al, bh, bl, ch, cl, dh, dl, es, si, ds, si, di, bp. То есть, если мы указали смещение к регистрам 6, то значение регистра ah будет в 6 – ой переменной, al – в 7-ой, bh – в 8-ой. Вот такая вот последовательность. Все значения регистров надо задавать перед выполнением команды вызова прерывания. После того, как эта команда выполнилась, можно прочитать значения регистров, которые это прерывание изменило.

Вот пример. Эти команды выведут на экран букву Ю, будем использовать функцию 0Eh прерывания 10h. Смещение к регистрам пусть будет равно 5, номер функции надо поместить в регистр ah, а он второй в списке, поэтому значение будем помещать в шестую переменную.

$(6)(~H0E)

Далее нам надо поместить букву Ю в регистр al, потому что эта функция выводит символ из этого регистра. Al первый в списке, поэтому кладём значение в пятую переменную.

```
$(5)('Ю')
```

Потом надо вызвать само прерывание. Тут всё просто.

```
~(9)(~H10,5)
```

В итоге программа выведет на экран букву Ю. Вот она вся:

```
$(6)(~H0E)
$(5)('Ю')
~(9)(~10H,5)
```

Правда, это всё можно было сделать простой командой $(1)('Ю'), но так интересней.

Есть ещё одна системная команда, которую тоже стоит изучить. Это команда прямой пересылки памяти, она позволяет пересылать не переменные, как команда присваивания, а байты оперативной памяти, что очень интересно. Вот синтаксис этой команды:

```
M(сегмент приёмник, смещение приёмник, сегмент источник, смещение источник, количество байт)
```

Да, таких больших команд у нас давно не было, но если разобраться, то тут всё просто. Начнём по порядку. Эта команда работает только с основной памятью ДОС, то есть только с 1 мб, а эта память, как известно, сегментирована, то есть любой байт в памяти имеет такой адрес: сегмент:[смещение]. Не будем углубляться в особенности сегментации памяти, это дело нескольких статей, лучше изучим особенности этой команды. Как мы видим, в ней два таких адреса – адрес источника и адрес

приёмника, то есть откуда будем брать и куда будем класть. Также там есть параметр «количество байт». С помощью него можно перемещать не один, а сразу целые цепочки байт, что бывает очень нужно в графических играх и других программах.

А теперь вопрос: как с помощью этой команды переместить какой-нибудь байт памяти в переменную? Например, значение байта 16:[64] переместить в третью ячейку. Для этого нужно знать расположение в памяти каждой переменной, то есть знать её сегмент и смещение. Это всё делается очень просто: для определения сегмента, в котором лежат переменные, нужно использовать оператор S, который возвращает номер этого сегмента, а смещение определяется тоже просто: нужно номер переменной умножить на два. Но не всё так просто: дело в том, что переменные в памяти занимают два байта, и чтобы переменная приняла значение того байта, нужно копировать в младшую часть переменной, то есть адрес переменной плюс один. Таким образом, наша задача будет выглядеть вот так:

```
M(S,7,16,64,1)
```

Почему ? Да потому что это 3*2+1, то есть адрес младшего байта третьей переменной. С остальным, думаю, понятно: 16-ый сегмент и смещение 64, а 1 означает количество копируемых байт. Надо только учесть, что значение старшего байта третьей переменной должно быть равно нулю, иначе результат будет неправильный, но если вы всё же копировали байт в переменную с неочищенным старшим байтом, то ничего страшного, его можно очистить с помощью команды AND вот таким методом:

```
$(3)(3$^~FF.)
```

Также с помощью этой команды можно перемещать просто переменные друг в друга. Вот, например, эта команда переместит значение переменной 10 в переменную 6:

```
M(S,12,S,20,2)
```

Цифра 2 в конце обозначает, что мы будем копировать два байта, потому что переменные имеют размер два байта.

А теперь напишем плохую программу. Очень плохую. Она затрёт адреса всех прерываний, что приведёт к невозможности всех операций ввода/вывода, что введёт программу в глубочайший ступор, из которого даже не любой резет сможет вывести. Сначала попытаемся разобраться с принципом работы этой программы.

Как известно, в самом начале памяти (0:[0]) находятся векторы прерываний, то есть адреса подпрограмм, обслуживающих эти прерывания. Например, мы вызвали 21h-тое (двадцать один хэтое) прерывание, процессор берёт с адреса 0:[84h] четыре байта (два на сегмент и два на смещение) и делает туда вызов процедуры. Эта структура показана на рис.0Ch. Теперь о том, как будет работать эта программа: она будет изменять эти адреса, и вместо того, чтобы вызвать процедуру обработки прерывания, будет вызываться всякая ерунда, которая либо не несёт полезного смысла, либо несёт, но не полезный. Как это реализовать? Да проще некуда: надо с помощью команды прямой пересылки

Рис.0Ch

памяти в область векторов прерываний переместить данные из какой-нибудь другой области. Это делается так:

M(0,0,675,4486,100)

Как мы видим, в этой команде переносится участок памяти, размером в 100 байт из какого-то случайного места в область векторов прерываний, тем самым затирая их. Затирается 100/4=25 прерываний, чего более чем достаточно, для остановки работы компьютера.

А теперь такой достаточно странный вопрос: а зачем мы писали эту программу? Неужели нельзя изучить эту команду на более мирном примере? Отвечаем: такую страшную программу мы написали для того, чтобы показать, что шутки с командами работы с системой могут окончиться очень плохо, а иногда даже плачевно, поэтому будьте осторожны.

2.8 Определения (Дефины)

Давным-давно мы с вами говорили о том, что позже научимся давать переменным имена. Думаю, это время пришло, и сейчас мы будем изучать определения, или дефины. Причем, с их помощью можно не только задавать имена переменным, но и менять названия команд, заменяя их на более понятные.

А если конкретно, то с помощью этой команды можно заменить одну часть строки другой строкой. Например, у нас есть команда вывода числа: $(2)(757). Первая часть этой команды нам не нравится, мы хотим заменить её на более понятную, например, Write, тогда команда будет выглядеть так: Write(757). Так вот, в начале программы мы пишем, что во всей программе строку Write надо заменить на $(2), и тогда к интерпретатору на выполнение будет поступать не Write, которого он не

понимает, а обычная команда записи во вторую ячейку. Теперь о том, как это всё записать. Записывается это так:

```
Define идентификатор=’замена’
```

В нашем случае это будет выглядеть так:

```
Define Write=’$(2)’
```

Имена переменным можно давать так же. Вот, например как можно дать третьей переменной имя X:

```
Define X=’3’
```

И тогда можно будет получать к ней доступ, записывая не 3$, а X$.

Все команды определений нужно писать в самом начале программы, даже до процедур, чтобы эти определения относились ко всему коду программы.

Для дефин есть некоторые ограничения. Задефинить нельзя некоторые команды, например h,b,define,uses. То есть, если вы напишите define h=’uu’, то интерпретатор выдаст ошибку.

Не запуская эту программу, попробуйте догадаться, что она будет делать?

```
Define much!=’+)’
Define I=’$(’
Define very=’2^2’
Define like=’2’
Define Stepler=’)(’
Program
```

```
I like stepler very much!
```

2.9 Библиотеки

Допустим, нам надо сделать очень большую программу, код которой слишком громоздкий для того, чтобы уместить его в одном файле; хочется разбить его на несколько маленьких. Или вы написали процедуры, которые могут использовать другие люди в своих программах, и хотите вынести их в отдельный файл. Что делать в этом случае? Сейчас мы это изучим.

В степлере можно использовать библиотеки – внешние файлы, содержащие процедуры, которые можно вызывать из основной программы. Это делается с помощью такой конструкции:

```
Uses 'имя библиотеки'
```

Вначале идёт команда uses, потом, в кавычках, идёт имя подключаемого файла, который имеет расширение .suf. После того, как файл подключился, можно вызывать процедуры из подключённого файла обычными командами вызова процедур, как будто эти процедуры находятся в основной программе. Команда uses пишется вначале программы сразу после команд define, если таковые есть.

Теперь давайте рассмотрим, как устроен подключаемый файл. Он состоит из нескольких частей:

1.Раздел Define. Здесь пишутся дефины. Раздел необязательный.

2.Раздел Uses. Здесь пишутся библиотеки, которые использует эта библиотека. Да-да, поддерживается вложенность. Раздел тоже необязательный.

3.Раздел процедур. Тут пишутся уже сами процедуры, которые можно вызывать из основной программы. Этот раздел обязательный.

Давайте теперь напишем простую библиотеку. Пусть она будет содержать абсолютно ненужные процедуры: Вывод Hello World, вывод числа и сложение двух чисел.

Первый раздел это раздел дефин. Мы не будем использовать дефины в этой библиотеки.

Второй пункт – библиотеки. Мы их тоже не будем использовать.

Третий пункт – процедуры. Их мы будем использовать, потому что это-основа библиотеки. Первая процедура – вывод Hello World. Эта процедура простая.

```
={HelloWorld}=
[Hello World]
**
```

Следующая процедура – ввод числа. Введённое число помещается в стек параметров.

```
={inputnum}=
P(1$)
**
```

Ну и последняя процедура – сложение двух чисел. Мы эту процедуру уже делали.

```
={summ}=
P(g^g+)
**
```

Напишем всё вместе:

```
={HelloWorld}=
[Hello World]
**
={inputnum}=
P(1$)
**
={summ}=
P(g^g+)
**
```

Вот и получилась простая библиотека. Сохраните её под названием useless.suf, и сейчас мы будем её использовать.

Вот простенькая программа, которая использует эту библиотеку. Она печатает ХеллоуВорлд с помощью процедуры, содержащийся в библиотеки.

```
Uses 'useless.suf'
Program
*<helloworld>
```

Вот так.

Внимание!

В интерпретаторе TOTOP нельзя использовать одинаковые метки в процедуре и в основной программе. В интерпретаторе ЛИНТ такой проблемы не наблюдется.

3.0 Алгоритмы

Вот мы и изучили с вами все команды степлера. Но вы сможете что-то на нём написать? Вряд ли. Степлер -это такой язык, который одним только перечнем команд не изучишь. Главное здесь- изучить основные алгоритмы работы на нём. Конечно, никто не заставляет следовать именно этим алгоритмам, вы можете изобрести свои и работать по ним, но если вам лень это делать, то можете использовать алгоритмы, описанные в книге, тем более, эти алгоритмы уже давно используются на практике и хорошо себя показали. Итак, приступим.

3.1 Условные блоки

Как известно, в степлере есть две команды контроля программы: условный переход и условный вызов процедуры. Но это только на первый взгляд. Если присмотреться, то окажется, что, используя основные команды перехода и вызова ,можно реализовать более сложные команды, аналоги условных блоков IF-THEN-ELSE.

Рассмотрим, как это можно сделать. Допустим, у нас есть группа команд, которые будут выполняться только при соблюдении какого-либо условия. Можно поступить таким образом: если условие совпадает, то продолжить выполнение программы, а если же не совпадает, то перейти на метку, расположенную после этого блока. Мы видим, что переход должен производиться тогда, когда условие НЕ совпадает. Это означает, что в команде безусловного перехода после условия надо поставить команду отрицания. Получаем такой код:

```
#(условие|)<then>
Команды блока
```

{then}

В этом коде команды, которые помечены как «команды блока» будут выполняться только тогда, когда выполняется условие.

То, что мы сейчас разобрали, называется условным блоком IF-THEN, а сейчас мы попробуем сделать условный блок IF-THEN-ELSE.

Что такое блок IF-THEN-ELSE? Это блок, в котором содержится два участка кода, и один выполняется, когда условие совпадает, а другой – когда не совпадает. Простейшим образом это можно сделать вот так:

```
#(условие|)<then>
Команды, которые выполняются, если условие верно.
{then}
#(условие)<else>
Команды, которые выполняются, когда условие неверно
{else}
```

Но в этом коде есть один недостаток: команды, которые находятся внутри условного блока, не должны менять само условие, так как если они его поменяют, то условие может неправильно заработать.

В таком случае можно использовать другую конструкцию:

```
#(условие|)<else>
Если совпадает условие
#<endif>
{else}
```

```
Если не совпвдает
{endif}
```

Тут мы видим, что если условие не выполняется, то производится переход на метку else, которая находится перед блоком команд, которые выполняются, когда условие не совпадает. Если же условие выполняется, то переход не происходит, выполняется соответствующий код, и происходит безусловный переход на конец условного блока.

Вот пример блока, который при условии, что третья переменная равна нулю, выводит «да», а если же не равно, то «нет»:

```
#(3$|)<else>
[Да]
#<endif>
{else}
[Нет]
{endif}
```

Вот так. Сейчас мы изучим ещё один способ создания условных блоков – с помощью процедур.

3.1.1 Условные блоки как процедуры

Сейчас мы научимся делать условные блоки с помощью процедур. Этот метод хорош тогда, когда условный блок повторяется и его надо вызывать несколько раз из разных мест программы. Суть его вот в чём: команды, выполняемые при условии, находятся внутри процедуры, а само условие пишется в команде вызова процедуры. Вот такая структура:

```
={uslblock}=
Условный блок.
**
… … …
*(условие)<uslblok>
```

Этот метод хорош, когда надо выполнить один и тот же блок в различных местах программы, так как эти команды пишутся всего один раз, тем самым не занимая места в программе.

В других случаях, когда этот блок располагается в программе один раз, лучше использовать тот тип условного блока, который мы изучали ранее.

Но мы знаем пока только один тип условий – если переменная равна нулю. Сейчас мы начнём главу, в которой будем изучать различные условия – больше, меньше, равно, и другие.

3.2 Условия

Да, жить с одним лишь типом условия – если равно нулю, очень не интересно. Поэтому мы сейчас научимся делать различные условия, которые можно использовать в своих программах.

3.2.1 Равно или неравно

Чтобы сделать условие- одно число равно другому- надо использовать свойство математики: если A=B, то A-B=0, а если равно нулю, то это то, что нам надо. Получается, чтобы сделать условие, что одно число равно нулю, надо из первого числа вычесть второе, и это будет условие:

```
Число1^Число2-
```

Вот пример условного перехода, который выполняется при условии, что третья переменная равна 10:

```
#(3$^10-)<label>
```

Давайте попробуем сделать более сложное условие. Пусть это будет условие, что 3$+4=5$. Вначале надо приравнять к нулю. Тогда получится вот такое выражение:

3$+4-5$=0. Переведём его на формат, понятный степлеру:

3$^4+^5$-. Получилось вот так.

Но теперь как сделать, чтобы было условие по неравенству двух чисел? А очень просто: надо в конце условия на равенство поставить знак отрицания. Вот, например, условный переход, который переходит, если третья ячейка неравна пятой ячейке.

```
#(3$^5$-|)<label>
```

Вот так.

3.2.2 Больше или меньше

А вот как нам сделать, чтобы сравнить два числа – больше одно другого или меньше? Для этого нам нужно снова вспомнить математику.

Мы знаем, что если A>B, то A-B>0, а если же A<B, то A-B<0. На языке операторов степлера это выглядит так: A^B-. Но как нам узнать, больше ли число нуля или меньше? А вот тут к нам на помощь приходит оператор определения знака числа, который превращает положительное число на вершине стека в 1, а отрицательное в -1. Поставим его после нашего выражения: A^B-!. Теперь, если первое число больше второго, то на вершине стека будет 1, если же меньше, то -1. Думаю, вы догадались, что дальше делать. Надо чтобы

при совпадении условия получался ноль, а это можно сделать либо прибавляя один, либо отнимая. Например, вот условие, что A>B:

```
A^B-!^1-
```

После оператора определения знака стоят операторы, которые отнимают 1. Это правильно, потому что при вычислении знака получается -1, а нам надо получить 0. Вот условие, «A<B»:

```
A^B-!^1+
```

Тут один прибавляется.

Тут всё просто, надо только запомнить, что всё наоборот: если «меньше», то плюс, а если «больше», то минус.

Вот пример. Переход производится, если значение третьей ячейки больше, чем пятой.

```
#(3$^5$-!^1-)<label>
```

3.2.3 Больше или равно, или меньше или равно

Думаете, это будет сложно? Ошибаетесь. Сделать условия таких типов очень легко, если мы умеем делать условия больше или меньше. Это делается так – «больше или равно» это «НЕ меньше», а «меньше или равно» это «НЕ больше». Вот так всё просто. Структура условия «Больше или равно»:

```
A^B-!^1+|
```

«Меньше или равно»:

```
A^B-!^1-|
```

Пример: вызов процедуры, если пятая переменная больше или равна пяти.

```
*(3$^5-!^1+|)<proc>
```

3.2.4 И, или, исключающие или

Допустим, нам надо вызвать процедуру не по одному условию, а сразу по нескольким, например, если третья ячейка равна двум, а пятая не равна семи. Что делать в этом случае? Можно сделать так, что первое условие вызывает одну процедуру, а та процедура по второму условию вызывает вторую. Но это не в нашем духе. Мы будем делать по-другому.

В степлере есть возможность собирать несколько условий в одно, чтобы общее условие выполнялось, когда выполняется хотя бы одно из них, либо когда выполняются оба, либо когда выполняется только одно. Это называется булевы операции, и сейчас мы научимся их делать.

Первое – если из двух условий выполняется хотя бы одно. Это называется логическое ИЛИ. Его можно реализовать так:

```
Усл1^Усл2*
```

Как мы видим, здесь производится умножение двух условий. Зачем это нужно? – подумаете вы. А вот зачем: когда условие выполняется, оно принимает значение 0, а если из двух множителей (здесь множители - это условия) один равен нулю, то и произведение (то есть общее условие) тоже равно нулю. Так-то вот.

Ещё одна операция над условиями – операция И. То есть общее условие будет выполняться только тогда, когда выполняются оба условия. Она имеет такой формат:

Усл1|^Усл2|*|

Зачем столько команд отрицания? – спросите вы. Дело в том, что когда условие выполняется, оно имеет значение нуля, но если же не выполняется, - любого другого числа. Так вот, если бы мы сделали так: Усл1^Усл2+, то было бы тоже самое, только вот не всегда бы оно работало. Например, если б нам надо было сделать такое условие: (3$=5)AND(4$=5). Мы бы написали так:

3$^5-^4$^5-+

Смотрим, что происходит: если обе переменные равны пяти, то всё верно, но если одна из них равна шести, а другая – четырём, то: (4-5)+(6-5)=-1+1=0!

Как?! Ни одно из условий не совпадает, а оно показывает, что оба верны? Это типичный глюк такой конструкции, поэтому я отказался от неё в пользу этой, более сложной, но верно работающей.

Ещё один тип логических операций, который не так часто используется, но всё равно, в некоторых моментах он просто необходим - исключающее или. Он имеет такую конструкцию:

Усл1|^Усл2|+^2%|

Не будем морочить вашу голову ужасающими подробностями того, как это работает, просто скажем, что это работает, и этого достаточно.

Рассмотрим несколько примеров условия:

(4>6)OR(6$=7)

Делаем так: берём первое условие:

4^6-!^1-

И второе:

6$^7-

И вставляем в шаблон:

4^6-!^1-^6$^7-*

Вот так всё просто. Пример посложнее:

((4$>=5$)OR(3=8$))XOR((7$=8)AND(8$<>5$))

Брррр. Кажется трудно? Нет, на самом деле всё просто.

Первое. Сделать четыре маленьких условия

4^5-!^1+|

3^8$-

7$^8-

8^5-|

Второе. Из четырёх маленьких сделать два побольше, основанных на OR и AND

4^5-!^1+|^3^8$-*

7$^8-|^8^5-||*|

Третье. Соединить два получившихся условия в одно, основанное на XOR

4^5-!^1+|^3^8$-*|^7$^8-|^8^5-||*||^2%|

Как видите, всё просто. А знаете, почему? Да потому, что пока вы дочитали книгу до этого места, в вашем мозгу появилось много нервных клеток, и благодаря этому, вы поняли всё здесь написанное, но если бы вы прочитали это, не читая до этого всю книгу, вы бы ни чего не поняли. Вот теперь вы наглядно видите, как степлер хорошо

тренирует мозг. Читайте дальше, и вы станете суперчеловеком! Удачи!

3.3 Модификация чисел

Допустим, нам надо решить такую задачу: при определённом условии выдать одно число, иначе другое число. Это можно сделать таким нудно-депрессивным методом: делать условные блоки и в них засовывать команды присваивания. Но это не наш метод, поэтому мы начинаем новый раздел – модификация чисел.

В этом разделе мы научимся превращать одно число в другое без использования условных переходов, а используя только операторы обратной польской. Это во много раз увеличивает скорость выполнения и уменьшает объём программы. Обязательно прочитайте эту главу. Серьёзно.

3.3.1 Числа 0 и другое число

Как мы знаем, условия представляют собой обычные числа – выполняется – ноль, не выполняется – другое число. Поэтому условия можно складывать, вычитать, умножать, делить с обычными числами. На этом и основан принцип модификации чисел. Простейший из них – при выполнении/не выполнении условия будут меняться местами ноль и какое-то число. Сейчас мы это рассмотрим.

Чтобы при выполнении условия получалось какое-то число A, а при не выполнении – 0, надо написать такой код:

```
Условие|^A*
```

Что мы там видим? Мы видим, что вначале командой отрицания результат условия инвертируется, то есть, выполняется – 1, не выполняется – 0. Потом этот

результат умножается на число А. Что тогда происходит? Когда условие выполняется и инвертируется, на выходе получается 1, который потом умножается это число А, а если число умножить на 1, то получится оно же, и поэтому на выходе получается число А. Что же происходит, если условие не выполняется? После инвертирования на выходе получится 0, так как до него был не ноль, так как условие не выполнялось. А если ноль умножить на любое число, то получится ноль. Так вот на выходе получается ноль.

Давайте рассмотрим пример. Он заносит в седьмую ячейку число 100, если третья ячейка раевна девяти.

```
$(7)(3$^9-|^100*)
```

А если наоборот, при выполнении условия ноль, а при невыполнении 1, то очень просто – надо проинвертировать условие, и получится вот так:

```
Условие||^A*
```

Пример, думаю, приводить не надо.

3.3.2 Два разных числа

А вот если нам надо, чтобы когда условие выполняется, было одно число, а если не выполняется – не ноль, а другое число.

Это можно сделать очень просто: надо написать команды модификации чисел 0 и другое число. Только чтобы то другое число было разностью двух чисел – из большего вынимаем меньшее. Потом надо прибавить меньшее число. Немного не понятно, сейчас разберёмся подробнее.

Вот структура.

Условие|^B-A*^A+

Разберём подробнее. Вначале мы будем модифицировать по условию разность этих чисел. Допустим, это числа 6 и 8. Тогда их разность будет 2. Делаем модификацию числа, если условие выполняется, то 2,если не выполняется, то 0, потом прибавляем меньшее – 6.

Получается, если выполняется условие, то получается 1*2+6=8, а если же не выполняется, то 0*2+6=6. Вот так.

При этом раскладе, получается, что при выполнении условия получается большее, а при невыполнении – меньшее. Если надо наоборот, то инвертируйте условие и будет ,как вам надо.

Вот пример, который кладёт в третью переменную 7, если условие выполняется, и 25, если не выполняется. Условие пусть будет такое, если пятая переменная равна девяти.

$(3)(5$^9-||^18*^7+)

Поскольку при выполнении условия нам нужно меньшее число, условие мы инвертируем.

Вот ещё один интересный пример, в котором модифицируется адрес переменной, в которую мы будем записывать число: если пятая переменная равна шести, то копировать в седьмую переменную восемь, а если же нет, то в 10-ую переменную 46. Вот решение, попробуйте разобраться с ним сами.

$(5$^6-|^3*^7+)(5$^6-|^38*^8+)

Как видите, сколько всего можно сделать с помощью обычной команды присваивания. Если бы мы использовали

условные блоки, то этот код выглядел бы вот так громоздко:

```
#(5$^6-)<else>
$(7)(8)
*<endif>
{else}
$(10)(46)
{endif}
```

3.3.3 Более двух чисел

А если у нас вот такая ситуация: при одном условии – одно число, при втором – второе, при четвёртом – четвёртое. Что нам тогда делать? Можно использовать такую конструкцию:

```
Усл1|^Число1*^Усл2|^Число2*^Усл3|^Число3*++
```

Здесь показаны три условия и три числа, но вы можете продолжать до бесконечности, главное помнить, что количество плюсов в конце на один меньше чем количество условий.

Как это всё работает? . Пишется несколько модификаций чисел в 0 и в 1, и все эти результаты сохраняются в стеке ОПЗ. Потом, все эти значения складываются и, при условии, что выполняется только одно условие, на вершине остаётся только одно число, то число, условие которого выполнилось. Те же условия, которые не выполнились, на это число не действуют, поскольку они нули, а нули на другие числа не действуют.

Рассмотрим пример. Допустим, если пятая переменная равна трём, то дать на выходе число 7, если же четырём, то шести, а если же восьми, то 122. Полученное число переместить в четвёртую переменную.

```
$(4)(5$^3-|^7*^5$^4-|^6*^5$^8-|^122*++)
```

Немного сложно, но, если разобраться, то всё понятно. Попробуйте разобраться сами.

3.3.4 Выражения вместо чисел

Но иногда бывают случаи, когда вместо чисел надо поставить какие-нибудь выражения, например, значение переменной или математическое выражение. Это тоже можно сделать. Лучше всего в таком случае использовать тот тип модификации чисел, который мы только что изучили – когда можно делать более двух чисел.

Ничем особенным использование выражений вместо чисел не выделяется. Всё точно также. Вот пример: если третья переменная равна шести, то в четвёртую переменную поместить значение десятой переменной, а если же не равна, то значение суммы шестой и седьмой переменных.

Рассмотрим это подробнее. Возьмём условие. Если третья переменная равна шести. Оно выглядит так: `3$^6-`, а выражение иначе выглядело бы так: `3$^6-|`, потому что оно обратное тому условию, а прямое и обратные условия одновременно быть не могут, получается, если из условия сделать обратное, то это будет «иначе».

Ладно, мы отвлеклись, продолжим. Ставим условие.

```
3$^6-
```

Далее ставим оператор отрицания и вверх, а потом выражение, которое должно выполняться. В нашем случае – 10$

```
|^10$
```

Потом пишем вверх, так как надо поместить значение в стек.

```
^
```

Потом надо условие «иначе», и другое выражение – у нас, сумма шестой и пятой переменной.

```
3$^6-||^5$^6$+*
```

Теперь надо всё сложить.

```
+
```

И дописать команду присваивания.

```
$(4)(3$^6-|^10$^3$^6-||^5$^6$+*+)
```

Вот так это всё выглядит.

3.4 Циклы

В самом начале, когда мы изучали команду условного перехода, мы сделали небольшую программку и назвали её циклической. Также мы сказали, что потом мы изучим несколько типов циклов. Вот сейчас мы это и сделаем.

Мы будем изучать три типа циклов.

Первый. С заранее известным количеством повторов.

Второй. С предусловием. Это когда вначале условие, потом тело. То есть, если условие не верно, тело цикла не выполняется ни разу.

Третий. С постусловием. Это когда вначале идёт тело цикла, а потом условие, цикл выполнится хотя бы один раз независимо от того, какое оно было до цикла.

Итак, приступим.

3.4.1 С известным количеством повторов

Не знаю, почему мы начали именно с этого типа циклов, хотя звучит он очень просто, но по структуре самый сложный из всех. Нарисуем его блок-схему (Рис.0Dh). Мы видим, что вначале устанавливаются значения начального счётчика, конца и шага. Что это такое? Сейчас разберёмся. У цикла, с известным количеством повторов есть специальная переменная, называемая счётчиком цикла. Она содержит число, которое во время работы цикла увеличивается или уменьшается, в зависимости от шага. Вот пример: начало отсчёта цикла – 4, конец – 10, шаг 2. Цикл сделает 4 прохода, и при этом счётчик будет принимать числа 4, 6, 8, 10.

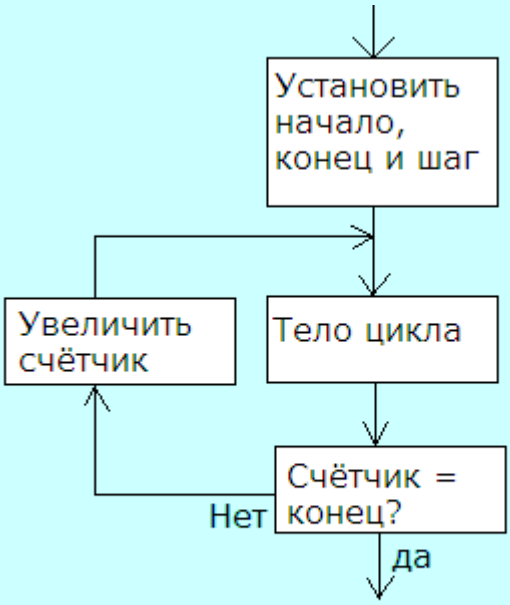


Рис. 0Dh Структура цикла

Теперь давайте поподробнее рассмотрим блок - схему этого алгоритма. Как мы видим,

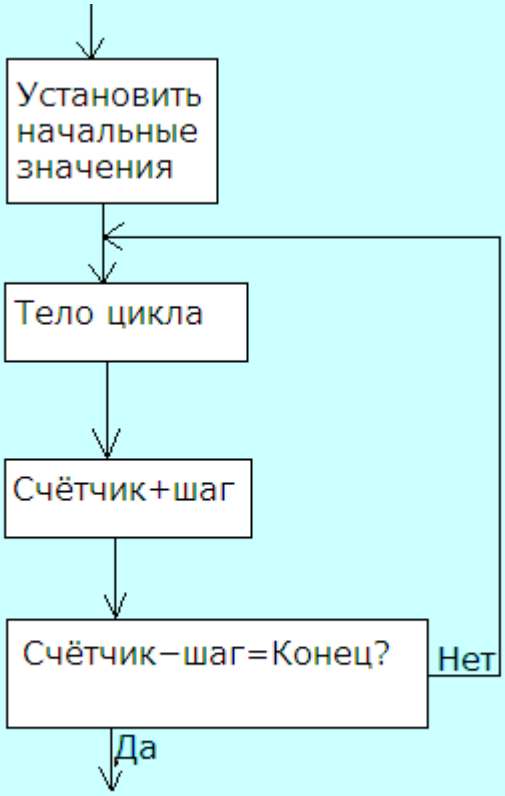


Рис. 0Eh Структура цикла.

алгоритм развёрнутый, а для степлера нужен линейный. Немного покумекав, его можно преобразовать (Рис.0Eh)

Давайте теперь по этой блок-схеме составим саму программу цикла.

Первое. Установить начальные значения. Здесь надо задать значение началу цикла, а для этого нужно определиться, какая переменная у нас будет счётчиком. Обозначим эту переменную СЦ, а начальное значение цикла НЦ.

```
$(СЧ)(НЦ)
```

Потом идёт метка.

```
{For}
```

Второе. Тело цикла. Так и напишем.

```
Тело цикла
```

Третье. Прибавить к счётчику цикла шаг. Это надо для того, чтобы цикл не стоял на месте, а продвигался. Не будет преувеличением сказать, что это одна из самых важных команд в цикле. ШГ – это шаг цикла.

```
$(СЦ)(СЦ$^ШГ+)
```

Четвёртое. Если Счётчик-Шаг<>Конец, то перейти на метку. Эта команда определяет, не кончился ли цикл. Но почему он проверяет не Счётчик<>Конец, а Счётчик-Шаг<>Конец? А потому, что перед этой командой стоит команда увеличения счётчика, а нам надо знать, какое было последнее значение счётчика на момент выполнения ТЕЛА ЦИКЛА, а это можно узнать ,если при сравнивании указывать, что счётчик на шаг меньше, чем было. Конец обозначим КЦ (Прям как в Кин-Дза-Дза). Вот сам код:

#(СЧ$^ШГ-^КЦ-|)<For>

Скомпонуем всё вместе.

$(СЧ)(НЦ)

{For}

Тело цикла

$(СЦ)(СЦ$^ШГ+)

#(СЧ$^ШГ-^КЦ-|)<For>

Вот так делается цикл с известным количеством повторов. Напишем программку, которая выводит все чётные числа от одного до ста. Началом будет служить первое чётное число – 2. Шагом будет тоже служить число 2, так как чётные числа располагаются через один, концом будет служить, явно, сто.

Program

$(3)(2)

{For}

$(2)(3$)

$(3)(3$^2+)

#(3$^2-^100-|)<For>

Если вы всё хорошо прочитали, то разобраться вам не составит труда.

А слабо сделать программу, в которой с клавиатуры вводятся начальное и конечное значение диапазона и потом подсчитывается и выводится сумма чисел этого диапазона?

3.4.2 С предусловием

Конструирование цикла, как всегда, начнём с блок-схемы. Она нарисована на рисунке 0Fh. Как мы видим, там есть условие, при выполнении которого выполняется тело цикла. После выполнения условия программа опять переходит к этому условию; если оно выполняется, то выполняется тело, потом опять к условию и так далее. Попробуем написать этот цикл на степлере.

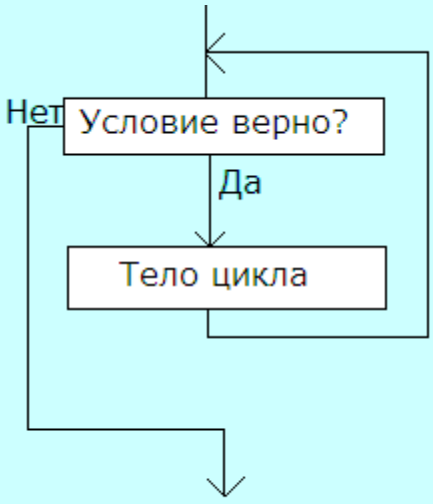


Рис. 0Fh. С предусловием

В начале мы видим, что выполнение программы передаётся на точку вначале программы. Это означает метку. Назовём её while1.

```
{while1}
```

Далее, мы видим переход; если условие не выполняется, значит, делаем команду перехода, и в условие ставим команду инвертирования. Переходим на метку while2

```
#(условие|)<while2>
```

Потом идёт тело цикла, а после него безусловный переход на метку while1, которая стоит в самом начале.

```
#<while1>
```

И в конце метка while2.

```
{while2}
```

Вот и всё. Теперь давайте соберём это всё вместе.

```
{while1}
```

#(условие|)<while2>

Тело цикла

#<while1>

{while2}

Теперь давайте изучим ещё один тип циклов – с постусловием.

3.4.3 С постусловием

В случае с циклами с предусловием может быть ситуация, когда тело цикла может выполниться ни разу. У циклов с постусловием это не так. Циклы с постусловием выполняются один и более раз, так что такой ситуации быть не может.

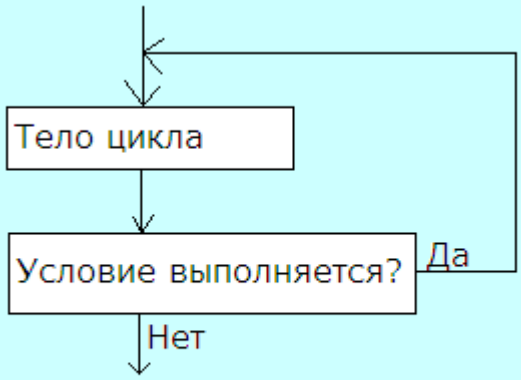


Рис.10h. С постусловием

Рассмотрим блок-схему этого цикла (Рис.10h). Как видим, она простая, намного проще цикла с предусловием, и тем более цикла с известным количеством повторов.

Вот его структура на степлере:

{Repeat}

Тело цикла

#(условие)<Repeat>

Напишем простенькую программку - пример. Пусть это будет ввод с клавиатуры ряда чисел, а потом вычисление их среднего арифметического. Конец ввода обозначить нулём.

Это можно сделать таким путём: все введённые числа запихнуть в массив, а потом циклом считать оттуда. Это очень нерациональный путь, должен быть другой. И он есть! Суть его такова: прямо при вводе чисел вычисляется сумма всех чисел, просто прибавляя введённое значение к уже готовой сумме. Также при вводе надо учитывать количество введённых чисел: ведь на их количество надо будет делить. Для создания цикла ввода будем использовать цикл с постусловием, так как хотя бы одно число должно быть введено. Условие будет такое: введённое число не равно нулю.

Надо также продумать, за что будет отвечать каждая переменная. В нашем случае пусть будет так: третья – сумма, четвёртая – количество, пятая – временная, содержит только что введённое число.

Начнём. Первая команда – Program.

```
Program
```

Далее надо начать цикл. Ставим метку.

```
{next}
```

Теперь начинается само тело цикла. В нём надо вывести предложение ввести число, ввести само число, прибавить его к сумме и увеличить количество введённых чисел.

```
[Введите число. 0-выход]
$(5)(2$)
$(3)(3$^5$+)
$(4)(4$^1+)
```

Потом команда выхода из цикла. Цикл продолжается, если введённое число не равно нулю.

```
#(5$|)<next>
```

Теперь сами вычисления. Вроде, всё просто, надо разделить сумму на количество, до вот одна загвоздка, надо делить не на количество, а на количество минус один, потому что ноль программа тоже приняла за число, а нам это абсолютно не нужно.

```
[Среднее арифметическое]
$(2)(3$^4$^1-/)
```

Вот такие вот дела. Вся программа:

```
Program
{next}
[Введите число. 0-выход]
$(5)(2$)
$(3)(3$^5$+)
$(4)(4$^1+)
#(5$|)<next>
[Среднее арифметическое]
$(2)(3$^4$^1-/)
```

Сейчас мы приступим к одной достаточно сложной, но интересной главе – массивы.

3.5 Массивы

Когда заходит речь о написании программ, которые работают с большим количеством данных, обязательно вспоминаются массивы, потому что как ни хороши переменные, много данных в них всё рано не засунешь.

Построение массивов в степлере основано на том принципе, что оператор $ берёт с вершины стека число и берёт ту переменную, номер которого есть это число.

3.5.1 Обычные массивы

Самый простой тип массивов – одномерные. На степлере они создаются так: в стек ОПЗ заносится номер элемента массива, прибавляется смещение и ставится оператор $, вызывающий переменную, если это нужно. Структура выглядит так:

```
Номер^Смещение+
```

А зачем нужно смещение? А вот зачем: если нужно что-то сохранить что-то в первой ячейке массива, то это число нарисуется символом на экране, что нас абсолютно не устраивает. Посмотрим на примеры. Допустим, у нас есть массив со смещением 8. Тогда команда, записывающая в третью ячейку число, двадцать, будет выглядеть так:

```
$(3^8+)(20)
```

А вот команда, которая выводит на экран значение десятого элемента массива:

```
$(2)(10^8+$)
```

В качестве номера элемента массива может быть не только просто число, но и переменная или выражение. Вот пример: на экран выводится элемент массива, номер которого хранится в третьей ячейке памяти.

```
$(2)(3$^8+$)
```

3.5.2 Доступ к массивам с помощью процедур

Есть ещё один способ создания массивов, немного более медленный, но более наглядный – с помощью

процедур. То есть, создать одну процедуру, которая будет записывать числа в массив, и другую, которая будет читать оттуда.

Напишем процедуру, заносящую значения в массив.

```
={setarray}= ;Параметры такие: [значение, номер]
$(8^g+)(g)
**
```

Как видите, тут всё просто. Используется обычная конструкция записи в массив, только берущая данные из стека параметров.

А вот процедура чтения из массива:

```
={readarray}=
P(8^g+$)
**
```

Тут тоже всё просто. Номер элемента массива берётся из стека параметров, и туда же возвращается его значение.

Надо только помнить, что эти процедуры для массивов со смещением 8. Если же вам нужно другое смещение, то измените это в процедуре.

3.5.3 Двумерные массивы

На самом деле двумерных массивов в степлере не существует. Мы просто берём одномерный массив и делаем к нему доступ, как к двумерному. Нечто похожее наблюдается в таблице Менделеева, так как это, по сути, одномерный массив, где каждый химический элемент имеет свой порядковый номер. В то же время эту таблицу можно рассматривать как двумерный массив, где каждый

элемент можно определить по двум параметрам – группе и периоду.

То же самое можно сделать и с любым другим двумерным массивом, да и не только двумерным, можно трёх-, четырёх-, и даже, пятимерным массивом. Сейчас мы научимся это делать.

Разлагать двумерный массив на одномерный мы будем по такой формуле: M(x,y)=A(x+y*h), где M – двумерный массив, A –его одномерный массив, x и y указывают на элемент двумерного массива, а h – ширина этого массива.

Доступ к этому массиву мы будем делать с помощью процедур. Вот процедура записи:

```
={writearray}=;[значение,x,y]
$(g^Ш*^g+^C+)(g)
**
```

Здесь Ш – ширина массива, а C – смещение.

Процедура чтения:

```
={readarray}=;[x,y]
P(g^Ш*^g+^C+)
**
```

Вот такие вот получились процедуры. Рассмотрим парочку примеров: записать в ячейку массива (5,6) число 48:

```
*[48,5,6]<writearray>
```

Вывести на экран значение элемента массива (8,9):

```
*[8,9]<readarray>
$(2)(g)
```

3.5.4 Динамические массивы

Что такое динамические массивы? Это массивы, которые могут менять свой размер во время выполнения программы. Сейчас мы посмотрим, как это можно реализовать на степлере.

Если мы говорим об одномерных (линейных) массивах, то ничего реализовывать не нужно, всё уже и так реализовано, так как ничто не ограничивает размер вашего массива, надо только следить за тем, чтобы он не затёр другие массивы или данные. С двумерными массивами всё немного сложнее: массив ограничен по ширине, и мы должны находиться в этих рамках.

Но что же нам мешает изменять ширину массива прямо во время работы программы? А ничего, именно на этом основан принцип работы динамических двумерных массивов. Сейчас мы рассмотрим, как это делается и для чего это нужно.

Делается это всё очень просто: создаём переменную, которая будет обозначать ширину массива. И, изменяя значение этой переменной, можно изменять ширину массива. Но делать это можно только до того, как мы записали туда данные. Так как если записать в двумерный массив данные, а потом изменить его ширину, то данные будут испорчены.

А теперь такой вопрос: зачем, собственно, это всё надо? А вот зачем: допустим, нам нужна программа, которая работает с большим объёмом данных, и мы не знаем, какого размера эти данные. Они могут быть очень маленькими, могут быть очень большими, но всё равно, программа должна с ними правильно работать. Тут как раз нам это и пригодится. С помощью какой-то процедуры

необходимо узнать размер этих данных, а потом установить размер массива, чтобы всё туда поместилось.

Теперь давайте напишем программу для примера. Пусть она требует ввода с клавиатуры ряда чисел, а потом, когда ввели ноль, выводит эти числа на экран.

Попробуем сделать её без рисования блок-схемы, всё-таки не маленькие уже.

Давайте решим, какие переменные и для чего мы будем использовать. Первые две переменные (в смысле третью и четвёртую) будем использовать в качестве временных, а массив будет располагаться, начиная с пятой переменной. Так какое же будет смещение у массива? 5? Нет, смещение будет 4, так как когда мы вычисляем положение элемента массива, мы к смещению прибавляем номер массива. А если поставить смещение 5, то первый элемент массива будет находиться в 5+1=шестой ячейке, а нам нужно в пятой, поэтому делаем смещение 4.

Доступ к массиву будем делать с помощью процедуры. Напишем процедуру чтения из массива.

```
={readarray}=
P(4^g+$)
**
```

И процедуру записи.

```
={setarray}=
$(4^g+)(g)
**
```

Вот теперь можно приступать к созданию самой программы. Как вы поняли, в ней есть два цикла. Первый

вводит числа в массив, а второй выводит из массива на экран. Надо решить, какого типа будут эти циклы. Первый цикл будет циклом с постусловием, то есть, вначале тело цикла, а потом условие. Второй цикл будет циклом с известным количеством повторов, так как при вводе массива мы будем считать количество введённых элементов. Итак, напишем первый цикл:

```
Program
$(3)(0)
{cycle1}
$(3)(3$^1+)
[Введите число. 0-выход]
$(4)(2$)
*[4$,3$]<writearray>
#(4$|)<cycle1>
```

Что мы здесь видим? Мы видим цикл, в теле которого происходит ввод числа с клавиатуры, занос этого числа в массив с помощью процедуры, увеличение счётчика и выход из цикла, если ввели 0. А такой вопрос: почему мы установили счётчик вначале в ноль, ведь у нас первый элемент массива – 1? А вот почему: увеличение счётчика происходит в начале тела цикла, и поэтому счётчик при первом выполнении тела будет равен 1.

Теперь надо написать второй цикл. Это цикл с известным количеством повторов, поэтому нам нужно знать такие параметры, как начало ,конец и шаг. Поскольку мы будем начинать выводить числа с начала, то начало отсчёта цикла будет равно одному. Шаг тоже будет равен одному, а конец будет равен не количеству элементов в массиве, а количеству элементов минус один,

потому что тот ноль, который мы вводили как признак конца, тоже был записан в массив, и, соответственно, если не уменьшить значение длины элементов массива на 1, то этот ноль тоже выведется, чего нам совершенно не надо.

Начнём писать цикл. Четвёртая переменная будет служить счётчиком.

```
$(3)(3$^1-)
$(4)(1)
{cycle2}
*[4$]<readarray>
$(2)(g)
$(4)(4$^1+)
#(4$^1-^3$-|)<cycle2>
```

Думаю, тут тоже всё понятно, есть цикл, в котором с помощью процедуры readarray читается значение нужного элемента массива, а потом выводятся на экран.

Напишем всю программу :

```
={readarray}=
P(4^g+$)
**
={writearray}=
$(4^g+)(g)
**
Program
$(3)(0)
{cycle1}
$(3)(3$^1+)
```

```
[Введите число. 0-выход]
$(4)(2$)
*[4$,3$]<writearray>
#(4$|)<cycle1>
$(3)(3$^1-)
$(4)(1)
{cycle2}
*[4$]<readarray>
$(2)(g)
$(4)(4$^1+)
#(4$^1-^3$-|)<cycle2>
```

Всем хороша эта программа, да вот только есть один недостаток: она слишком длинная. 21 строка для такой простой программы - это слишком много. Попробуем сократить количество строк. На что можно обратить внимание? Во-первых, есть две процедуры, которые используются всего один раз. Такого следует избегать, потому что процедуры занимают лишние строки и время на их вызов и возврат из них. Удалим процедуры, а те действия, которые они производят, переместим в сам код программы.

```
Program
$(3)(0)
{cycle1}
$(3)(3$^1+)
[Введите число. 0-выход]
$(4)(2$)
$(4^3$+)(4$)
```

```
#(4$|)<cycle1>
$(3)(3$^1-)
$(4)(1)
{cycle2}
$(2)(4^4$+$)
$(4)(4$^1+)
#(4$^1-^3$-|)<cycle2>
```

Итак, у нас уже 14 строчек, но, наверное, можно сократить ещё. Смотрим на строчку уменьшения количества элементов массива перед вторым циклом. Что интересно, её можно убрать, но не совсем, а это уменьшение переписать в значение конца цикла, то есть, вместо `#(4$^1-^3$-|)<cycle2>` написать

`#(4$^1-^3$^1--|)<cycle2>`

Но и это можно сократить. Превратим это выражение на ОПЗ в обычное математическое выражение. Оно будет выглядеть так: (4$-1)-(3$-1). Раскроем эти скобки. 4$-1-3$+1, сократим и получим: 4-3. Получается, что последнюю команду можно сократить до вида:

`#(4$^3$-|)<cycle2>`. Вот вся программа после всех сокращений.

```
Program
$(3)(0)
{cycle1}
$(3)(3$^1+)
[Введите число. 0-выход]
$(4)(2$)
```

```
$(4^3$+)(4$)
#(4$|)<cycle1>
$(4)(1)
{cycle2}
$(2)(4^4$+$)
$(4)(4$^1+)
#(4$^3$-|)<cycle2>
```

Как видите, мы сократили размер программы с 21 до 13 строк, при этом не теряя ни одной функции. Этот процесс называется оптимизацией, и сейчас мы углубимся в дебри этого увлекательного занятия.

3.6 Оптимизация

Иногда бывает такое: программа написана, но либо работает слишком медленно, либо её размер слишком большой. В этом случае нужно применить оптимизацию – процесс изменения программы для улучшения её эффективности. Но иногда двух зайцев одновременно поймать не удаётся – уменьшение размера ведёт к замедлению работы программы, а ускорение работы – к увеличению объёма. Но всё равно – оптимизация это хорошо, и её можно применить к практически любой программе.

3.6.1 Удаление процедур

Этот метод оптимизации нам уже знаком. Его мы применяли, когда оптимизировали программу работы с массивами. Но всё равно, его надо изучить глубже.

Так что такое оптимизация программы путём удаления процедур? Это когда мы удаляем процедуру, а на все места её вызовов подставляем содержимое этой

процедуры. Зачем нужна эта кухня? А вот зачем: если во время работы программы эта функция вызывается всего один раз, то это преобразование несущественно, но всё равно, сократит размер программы, так как операции работы со стеком параметров и стеком возвратов требуют достаточно много времени. Таким образом, подставляя содержимое процедуры на место её вызова, мы экономим время на:

1.Передачу параметров

2.Вызов процедуры

3.Забор параметров

4.Передач результата

5.Возврат

6.Забор результата

Как видите, сколько лишних операций не будет выполняться, если мы сделаем это преобразование. А если процедура вызывается не один, а несколько раз? Ну, если процедура не такая большая и вызывается не так очень много раз, то, лучше сделать это преобразование.

Однако после выполнения этой оптимизации у программы будет менее читаемый исходный код, потому что не будет так ясно, что сейчас происходит.

Пример удаления процедур рассматривать не будем, так как мы уже разбирали его в программе работы с массивом.

3.6.2 Удаление лишних строк

Этот метод похож на тот, что мы только что изучали, за тем исключением, что удаляются не вызовы процедур, а

обычные команды, результат выполнения которых используется один или несколько раз.

А вообще, суть этой оптимизации в том, чтобы засунуть часть одного числового выражения в другое. Вот пример:

```
$(5)(7$^8$-|^3-!*^9-^5^4$++*)
$(6)(5$^7-)
```

Можно скомбинировать два этих выражения, поставив на то место, где стоит 5$, выражение из первой строки.

Получится одна длинная команда:

```
$(6)(7$^8$-|^3-!*^9-^5^4$++*^7-)
```

Но выполняется она быстрее, чем две аналогичные.

Ещё один способ избавиться от лишних строк – использовать модификацию чисел вместо условных блоков. Это намного ускоряет работу и уменьшает количество строк кода.

Вот пример использования оптимизации этого типа. Возьмем такой код: если значение в пятой ячейке больше десяти, то переместить это значение в ячейку, номер которой находится в седьмой переменной, а если же меньше или равно, то в следующую.

Традиционным способом, с помощью условных блоков, это можно сделать так:

```
#(5$^10-!^1-|)<else>
$(7$)(5$)
#<endif>
{else}
```

```
$(7$^1+)(5$)
{endif}
```

Как видим, достаточно много строк и, кажется, что сократить больше нечего. Но на самом деле есть. Надо использовать команды модификации чисел так, чтобы из-за условия модифицировался адрес, в который мы будем перемещать значение пятой переменной, и прибавлять или не прибавлять в зависимости от условия. Если не выполняется – прибавляем один, а если выполняется - не прибавляем.

Берём условие:

```
5$^10-!^1-|
```

Далее берём шаблон команды модификации чисел.

```
Условие|^A*
```

Вместо A ставим один, поскольку нам надо прибавлять это число, и видим, что происходит умножение на один, а если умножить на один, то получится то же самое число. Поэтому ^A* можно опустить. Также надо, чтобы при выполнении условия получался ноль, а не один, поэтому инвертируем условие, ставя в конце команду отрицания. Шаблон получается такой:

```
Условие||
```

Подставляем наше условие в шаблон.

```
5$^10-!^1-|||
```

В конце видим три команды отрицания. Их можно заменить на одну, так как они выполняют то же самое. И вообще, если стоит более двух подряд команд отрицания, то можно отнимать два до тех пор, пока не будет два или меньше.

Так вот, мы получили такую команду модификации. Как видим, она ничем ни отличается от условия, поэтому можем сделать вывод, что это условие и есть сама команда модификации.

Теперь надо поставить эту конструкцию в команду присваивания. Она будет находиться в поле приёмника и прибавляться к седьмой переменной. А в поле источника должна находиться пятая переменная. Всё будет выглядеть так:

```
$(7$^5$^10-!^1-|+)(5$)
```

Вот видите, как можно сократить код программы, используя оптимизацию. Этот код делает то же самое, что и тот, который использует условные блоки, но при этом намного короче и выполняется быстрее.

3.6.3 Меньшее количество вызовов функций

Главный принцип этой оптимизации заключается в том, что если значение функции вычисляется много раз, и при этом с одинаковым параметром, то лучше однажды вычислить её значение, поместить ответ в какую-то переменную и потом использовать эту переменную, а не вызов функции. Это даёт намного больший прирост скорости, так как бывают функции, которые содержат циклы и поэтому выполняются очень долго.

Например, мы делаем программу, которая производит вычисления из корня какого-то числа. Так вот, лучше вначале узнать значение этого корня, а потом, не вычисляя его при каждой надобности, использовать его в выражениях. На примере эту оптимизацию мы рассмотрим чуть позже.

3.6.4 Замена процедур массивами

Однако, иногда для работы программы нужен целый диапазон значений какой-либо функции, которая выполняется достаточно долго. Эту задачу можно решить довольно интересным способом: перед выполнением программы записать значения данной функции в массив, а

потом уже не вызывать эту функцию, а читать уже вычисленное значение из массива.

Вы удивитесь, но этот метод широко используется в повседневной жизни. Например, когда нам надо узнать синус какого-то числа, мы не вычисляем его с помощью очень сложных формул, а смотрим в таблицу, где эти значения уже вычислены.

Также можно читать значения функций из файла, предварительно загрузив его в память, поскольку чтение из файла – достаточно медленное занятие.

3.7 Поиск

В этой главе давайте рассмотрим методы поиска в одномерном массиве. Будем изучать два типа поиска: поиск в неупорядоченном массиве и поиск в упорядоченном (отсортированном) массиве.

3.7.1 Поиск в неупорядоченном массиве

Поиск в неупорядоченном массиве очень прост. Это обычный цикл с постусловием, который выполняется до тех пор, пока не найдётся искомый элемент или не кончится массив.

Давайте нарисуем блок-схему(рис 11h)

Мы видим, что вначале ставится метка, что элемент не найден. Это делается устанавливанием номера найденного элемента в -1. Это достаточно хороший способ, потому что элемента с таким номером нет. Далее, идёт цикл, который проверяет каждый элемент массива, и, если он равен искомому, то дать переменной, обозначающей номер найденного элемента, значение текущего элемента и выйти из цикла.

Теперь давайте это рассмотрим на практике.

Пусть счётчиком цикла будет третья переменная, а четвёртая переменная обозначает номер найденного элемента, а в пятой будет находиться искомый элемент. Массив будет находиться в памяти со смещением 10.

Начнём. Первое. Найденный элемент равен -1. Значит, надо четвёртой переменной присвоить значение-1.

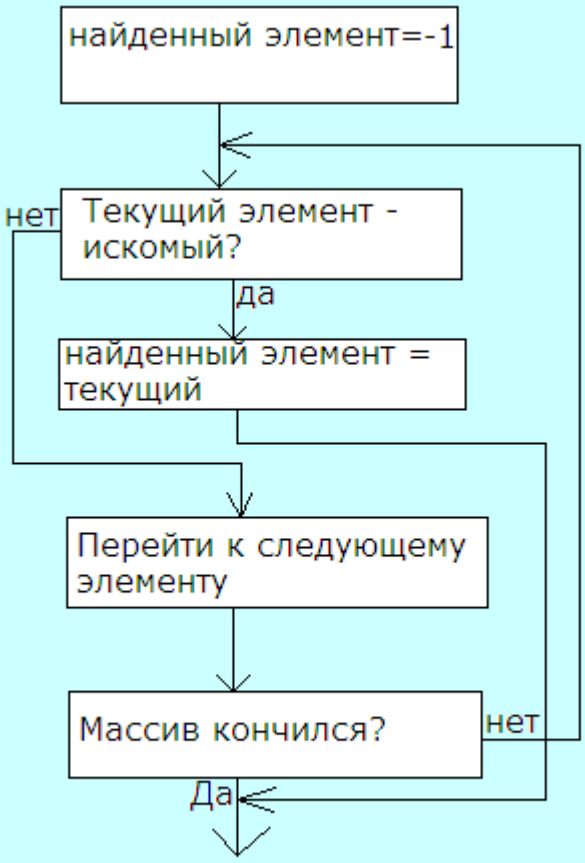


Рис.11h Поиск элемента

```
$(4)(0^1-)
```

Далее идёт метка. Назовём её cycle.

```
{cycle}
```

Второе. Безусловный переход. Условие будет такое: переход осуществляется, если текущий элемент не равен искомому.
Будем использовать правила, что если A<>B, то A-B<>0. Метка, на которую мы будем переходить, будет называться find.

```
#(10^3$+$^5$-|)<find>
```

Третье. Найденный элемент=текущий. Надо присвоить ячейке, обозначающей текущий элемент, номер текущей ячейки.

```
$(4)(3$)
```

Далее идёт переход на метку, находящейся в конце программы. Назовём её endfind.

```
#<endfind>
```

Потом находится метка, на которую мы переходили вначале. Метка find.

```
{find}
```

Четвёртое. Перейти к следующему элементу. Увеличиваем значение текущего элемента.

```
$(3)(3$)
```

Пятое. Если массив кончился, то закончить, иначе перейти к началу. Здесь нам надо знать, какой элемент массива был просмотрен последним, а если мы возьмём просто ячейку, обозначающую текущий элемент, то у нас там будет значение текущей ячейки+1, так как мы его увеличивали. Поэтому нам надо уменьшить значение, чтобы узнать последний просмотренный элемент. Метку, на которую он будет переходить, мы обозначили cycle, а в шестой переменной находится размер массива.

```
#(3$^1-^6$-|)<cycle>
```

И в конце метка endfind.

```
{endfind}
```

Ну вот и всё, теперь можно написать весь код:

```
$(4)(0^1-)
{cycle}
#(10^3$+$^5$-|)<find>
$(4)(3$)
#<endfind>
```

```
{find}
$(3)(3$)
#(3$^1-^6$-|)<cycle>
{endind}
```

Эта часть кода будет совершать поиск элемента в массиве, и если он находит его, то прекращает поиск и указывает номер найденного элемента, а если же не находит, то устанавливает номер найденного элемента в -1.

Попробуйте написать программку, в которой вы с клавиатуры вводите массив и элемент, который надо найти, а программа ищет его в массиве, и если находит, то выдаёт его номер.

3.7.2 Поиск в упорядоченном массиве

Тот поиск, который мы только что изучили, хорош для любых массивов – для отсортированных и неотсортированных, но у него есть один недостаток: он медленный, приходится просматривать все элементы, пока не найдётся нужный. Если искать в маленьких массивах, то это сильно не мешает, но если же производить поиск в длинных массивах, то это серьёзно затрудняет работу. Сейчас мы изучим ещё один способ поиска, который работает только в отсортированных массивах, но очень сильно ускоряет процесс поиска.

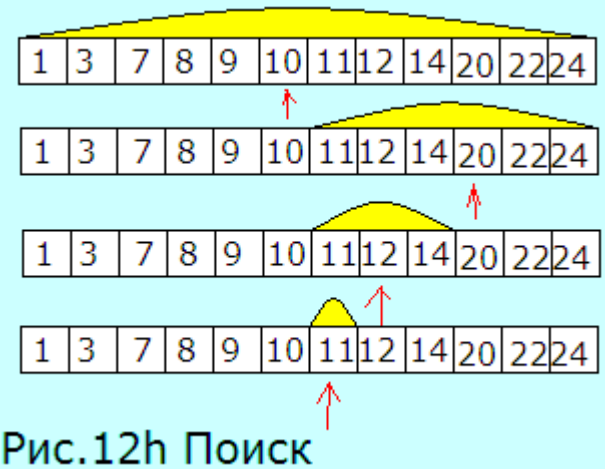


Рис.12h Поиск

В чём заключается суть этого поиска? В массиве выделяется область, в которой находится этот

элемент, а потом эта область постоянно сужается и в конце концов в ней не находится элементов вообще, если этого элемента нет, либо находится найденный элемент. Каким же образом происходит сужение? А вот таким: берётся серединный элемент этой области, и если он больше искомого, то правая граница области устанавливается в номер текущего элемента минус один, а если же меньше, то левая граница устанавливается на номер текущего элемента плюс один. Поиск числа 11 в отсортированном массиве показан на рисунке 12h. Что мы там видим: вначале областью, в которой находится элемент, является весь массив, потом берётся средний элемент (10) и сравнивается в искомым. Поскольку он меньше искомого, то граница устанавливается на его номер плюс один, то есть на следующий. Итак, мы знаем, что искомый элемент находится в области, которая занимает вторую половину массива. Берём оттуда серединное число (20) и сравниваем его и искомым. 20 больше одиннадцати, поэтому устанавливаем правую границу на номер текущего элемента минус один, то есть на предыдущий номер. Итак, у нас уже есть область из трёх чисел, берём оттуда число посередине (12), сравниваем, смещаем границу и видим, что область состоит всего из одного элемента, и он – тот самый искомый. Вот так всё интересно.

А теперь давайте весь этот ужас реализуем на степлере. Вначале предлагаю нарисовать блок-схему. (рис. 13h)

Ололо, давно у нас не было такого сложного алгоритма, но ничего, всё норм. Сначала давайте разберёмся, что это за буковки т.э и п.г, да некоторые ещё в скобках.

Итак, к.м – конец массива, л.г – левая граница, п.г – правая граница, н.э – нужный элемент, т.э – текущий элемент, а если оно в скобках, это значит, что мы используем не само значение, а элемент массива с этим значением. Начнём потихоньку переводить эту блок-схему в программу. Пусть в третьей переменной находится т.э, в четвёртой н.э, в пятой л.г, в шестой п.г, в седьмой к.м, а смещение у массива – семь ячеек.

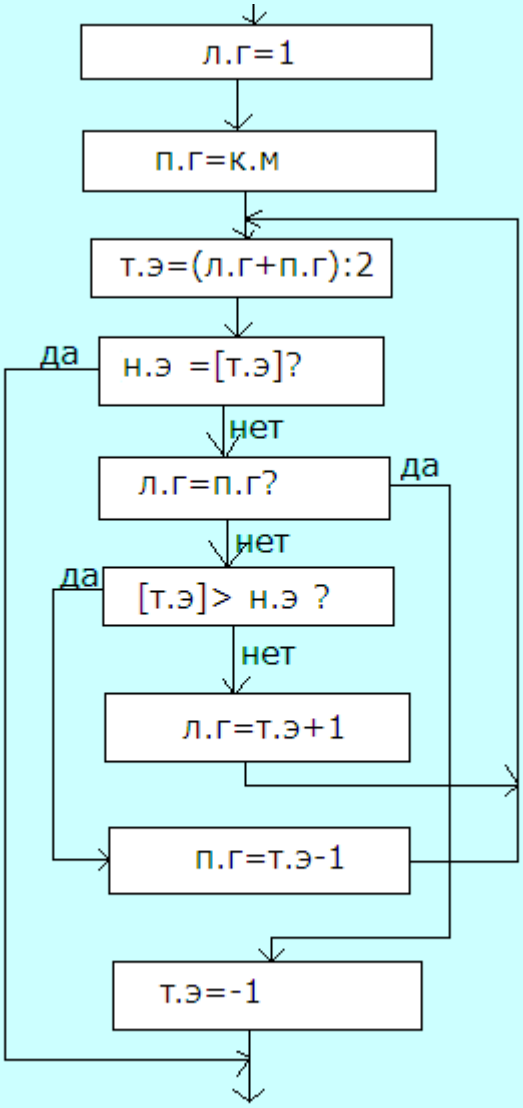


Рис.13h Алгоритм поиска

По алгоритму вначале у нас идёт установка границ области. Сделаем это.

```
$(5)(1)
$(6)(7$)
```

Потом идёт метка, и выражение, которое высчитывает серединный элемент, просто вычисляя среднее арифметическое между левой и правой границей области.

```
{cycle}
$(3)(5$^6$+^2/)
```

Потом идёт три условных перехода, которые определяют, что делать в текущей ситуации. Первый из них переходит,

если значение текущего элемента массива равно искомому (нужному) элементу, то есть, если элемент найден. Второе условие срабатывает, если правая граница равна левой границе, то есть размер области – одна ячейка. А если честно – искомого элемента в массиве нет вообще, поскольку размер области одна ячейка, и эта ячейка – не искомый элемент, так как если бы это была она, то сработал бы переход в предыдущем пункте, и эта команда не выполнялась бы. А третий условный переход срабатывает, если значение текущего элемента массива больше искомого элемента. Напишем эти команды теперь на степлере. Первый условный переход :

```
#(4$^3$^10+$-)<najden>
```

Второй:

```
#(5$^6$-)<nenajden>
```

Третий

```
#(3$^10+$^4$-!^1-)<bolshe>
```

Далее идёт такая ситуация : мы выполняем действие, которое должны выполнять, если текущий элемент меньше нужного. Почему? А потому, что программа доходит до этого места только в том случае, если оно так и есть, то есть, если текущий элемент меньше искомого. А если текущий элемент меньше найденного, то надо установить левую границу в номер текущего элемента плюс один.

```
$(5)(3$^1+)
```

Потом переходим на метку вначале, то есть, cycle.

```
#<cycle>
```

А теперь, мы видим, что управление переходит на действие и переходит в начало на метку cycle. Это надо

написать так: вначале метка, на которую передаётся управление, потом само действие, и в конце, переход на метку cycle.

Это действие будет выполняться, если значение текущего элемента больше искомого, поэтому надо установить значение правой границы в номер текущего элемента минус один.

```
{bolshe}
$(6)(3$^1-)
#<cycle>
```

Далее, находится метка, действие на которую передаётся, если элемент не найден, в этом действии надо установить значение найденного элемента в -1.

```
{nenajden}
$(3)(0^1-)
```

Ну и в самом конце, метка, обозначающая конец, или, если элемент найден:

```
{najden}
```

Вот и всё. Давайте запишем весь код.

```
$(5)(1)
$(6)(7$)
{cycle}
$(3)(5$^6$+^2/)
#(4$^3$^10+$-)<najden>
#(5$^6$-)<nenajden>
#(3$^10+$^4$-!^1-)<bolshe>
$(5)(3$^1+)
```

```
#<cycle>
{bolshe}
$(6)(3$^1-)
#<cycle>
{nenajden}
$(3)(0^1-)
{najden}
```

Да, код достаточно длинный. Намного длиннее, чем метод обычного поиска, но он намного быстрее, давайте посмотрим, на сколько.

Если искать обычным, последовательным поиском, то для того, чтобы найти элемент, требуется столько циклов, сколько элементов в массиве. То есть, если в массиве 100 элементов, то, в худшем из вариантов потребуется 100 проверок, чтобы найти нужный элемент. В лучшем случае потребуется всего одна проверка, это если искомый элемент в массиве стоит первый.

А если же искать этим методом, то для того, чтобы вычислить время, которое потребуется для поиска в худшем случае, надо немного подумать. Мы знаем, что при каждой проверке проверяемый участок уменьшается в два раза, и поиск считается законченным, если проверяемый участок равен одному элементу. Таким образом, количество проверок равно количеству усечений массива в два раза до тех пор, пока его длина не станет 1. Можно вывести такую формулу: N : 2 : 2 : 2 … : 2 =1, где N – длина массива.

Итак, нам надо узнать количество делений на два. Можно изменить формулу: $N\ (:2)_k = 1$, изменим ещё: $N : 2^k = 1$ или $N = 2^k$. Так вот, k – и будет тем самым

количеством сравнений. Для ста это будет примерно 100 ~~ 128 = 2^7, то есть семи сравнениям! Вот видите, во сколько раз этот поиск быстрее обычного, но, к сожалению, он работает только на отсортированных массивах, на неотсортированных приходится применять обычный медленный поиск.

А теперь давайте напишем программу, которая просит ввести с клавиатуры отсортированный массив, искомый элемент, а потом находит его и говорит его номер.

Напишем команды ввода с массива с клавиатуры.

```
Program
$(3)(0)
{cycle1}
$(3)(3$^1+)
[Введите число. 0-выход]
$(4)(2$)
$(10^3$+)(4$)
#(4$|)<cycle1>
$(3)(3$^1-)
```

Размер массива хранится в третьей переменной, а нам надо, чтобы был в седьмой, поэтому переместим.

```
$(7)(3$)
```

Введём искомое число.

```
[Введите число, которое надо найти]
$(4)(2$)
```

Далее надо написать сам участок кода, отвечающий за поиск

```
$(5)(1)
$(6)(7$)
{cycle}
$(3)(5$^6$+^2/)
#(4$^3$^10+$-)<najden>
#(5$^6$-)<nenajden>
#(3$^10+$^4$-!^1-)<bolshe>
#<cycle>
{bolshe}
$(6)(3$^1-)
#<cycle>
{nenajden}
$(3)(0^1-)
{najden}
```

И теперь, если элемент не найден, надо вывести соответствующую надпись, а если найден, то вывести его номер.

```
#(3$^0^1--|)<else>
[Элемент нее найден]
#<endif>
{else}
[Элемент найден. Его номер]
$(2)(3$)
{endif}
```

Напишем теперь всё вместе.

```
Program
```

```
$(3)(0)
{cycle1}
$(3)(3$^1+)
[Введите число. 0-выход]
$(4)(2$)
$(10^3$+)(4$)
#(4$|)<cycle1>
$(3)(3$^1-)
$(7)(3$)
[Введите число, которое надо найти]
$(4)(2$)
$(5)(1)
$(6)(7$)
{cycle}
$(3)(5$^6$+^2/)
#(4$^3$^10+$-)<najden>
#(5$^6$-)<nenajden>
#(3$^10+$^4$-!^1-)<bolshe>
$(5)(3$^1+)
#<cycle>
{bolshe}
$(6)(3$^1-)
#<cycle>
{nenajden}
$(3)(0^1-)
```

```
{najden}
#(3$^0^1--|)<else>
[Элемент не найден]
#<endif>
{else}
[Элемент найден. Его номер]
$(2)(3$)
{endif}
```

Огого, какая программа получилась. Давайте её оптимизируем. Итак, что мы имеем? Когда мы вводим массив, мы из его размера вычитаем 1, перемещаем в седьмую, а потом в шестую ячейку. А нельзя ли сразу переместить в шестую ячейку третью ячейку минус один? Можно! Одно уже оптимизировали. Далее. В конце стоит условный блок, определяющий, нашли ли мы элемент или нет. Обратите внимание, вначале мы условным переходом устанавливаем этот минус один, а потом смотрим, не равен ли он минус одному? Получается, проверяем одно и то же два раза. Это нужно убрать. Можно поступить так: сразу после перехода на метку nenajden - а на неё производится переход, только если элемент не найден - пишем команду вывода сообщения и безусловный переход на конец программы, а после метки najden пишем сообщение о том, что найден, и выводим номер. Оптимизированная программа выглядит так:

```
Program
$(3)(0)
{cycle1}
$(3)(3$^1+)
```

```
[Введите число. 0-выход]
$(4)(2$)
$(10^3$+)(4$)
#(4$|)<cycle1>
[Введите число, которое надо найти]
$(4)(2$)
$(5)(1)
$(6)(3$^1-)
{cycle}
$(3)(5$^6$+^2/)
#(4$^3$^10+$-)<najden>
#(5$^6$-)<nenajden>
#(3$^10+$^4$-!^1-)<bolshe>
$(5)(3$^1+)
#<cycle>
{bolshe}
$(6)(3$^1-)
#<cycle>
{nenajden}
[Элемент не найден]
#<end>
{najden}
[Элемент найден. Его номер]
$(2)(3$)
{end}
```

Таким образом, в процессе оптимизации программа была сокращена с 34 до 28 строк. А сейчас мы будем изучать сортировку массива.

3.8 Сортировка

Сортировка массива – одна из наиболее изучаемых областей информатики, и на текущий момент существует большое множество алгоритмов сортировки, один лучше другого. Некоторые сортировки лучше для маленьких массивов, другие для больших, третьи для сортировки таблиц, четвёртые – ещё для чего-то.

Мы же будем изучать один метод – сортировка пузырьком. Этот метод достаточно простой и хорошо подходит для небольших программ. Давайте его рассмотрим.

Рис.14h -Сортировка пузырьком

Сортировка пузырьком происходит следующим образом: сравниваются два соседних элемента, и если первый больше второго, то они меняются местами. Такая процедура производится с каждым элементом массива в количестве «размер массива минус один». Вот на рисунке 14h изображена сортировка массива, состоящего из четырёх элементов.

На рисунке мы видим исходный неотсортированный массив :4,8,6,2. Далее проверяем первый и второй элементы, поскольку они стоят в правильной последовательности (то есть по возрастанию), то менять их не стоит. Далее проверяем второй и третий элементы. Они стоят в неправильной последовательности, поэтому мы их меняем, то есть вначале было 8,6, а стало 6,8. Проверяем третий и четвёртый элементы, они тоже в неправильной последовательности, поэтому меняем их местами.

Мы сделали один проход массива. Но, как видно, он не отсортирован. Делаем второй проход, во время которого мы ещё немного отсортируем его, а потом третий, после которого массив уже будет полностью отсортирован. Итак, для четырёхэлементного массива нужно три прохода. Для пятиэлементного – 4, а для N-элементного N-1. Таким образом, чтобы отсортировать массив, надо сделать N проходов, и в каждом проходе N-1 сравнений, то есть всего (N-1)*(N-1)= $(N-1)^2$ = N^2-2N+1 сравнений.

Теперь давайте нарисуем блок – схему и подойдем к этому аналитически.

Итак, что мы здесь должны увидеть? Во-первых, цикл, который должен считать количество проходов. Он будет считать от 1 до N-1. Потом ещё один цикл, который указывает текущую сравниваемую пару чисел. Он тоже считает от 1 до N-1, поскольку он сравнивает N-ый элемент и N+1-ый, а поскольку у нас в массиве N элементов, то ,чтобы он не брал N+1-ый элемент, надо считать до N-1.

Итак, это будет два цикла, причём один вложен в другой. А что будет внутри этих циклов? Будет условие,

которое сравнивает текущее и следующее число, и если следующее - меньше, то меняет их местами.

Вот и всё. Теперь можно нарисовать блок-схему(Рис.15h). Вроде бы там всё понятно – два вложенных цикла и внутри простой алгоритм переворачивания по условию. Да, это можно сделать стандартным путём, создав два вложенных цикла с метками и условиями. Но можно пойти и другим путём.

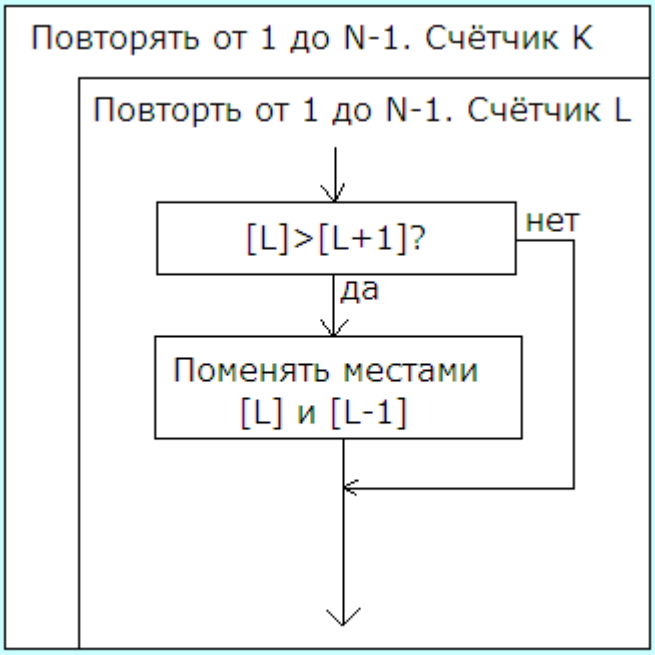


Рис.15h Сортировка массива

Как мы знаем, в степлере есть операция вычисления остатка от деления, она обозначается %. Кроме нахождения остатка от деления у неё есть ещё одно предназначение: делать вот такие пилообразные изменения чисел. Какие это пилообразные? – спросите вы. А вот такие – когда мы сортируем массив, то номер проверяемой ячейки изменяется так – 1,2,3,1,2,3,1,2,3. Если эту последовательность представить в виде графика, то он будет похож на пилу, а особенно на тот тип пилы, которым пилят параллельно древесным волокнам. (Рис.16h)

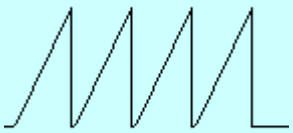


Рис.16h

Так вот, давайте посмотрим, какие значения будет принимать функция остатка от деления на три: 0%3=0, 1%3=1, 2%3=0, 3%3=0, 4%3=1… То есть 0, 1, 2, 0, 1, 2, 0, 1, 2. Как видите, эта последовательность отличается от предыдущей только тем, что каждое значение на один

меньше. Так надо прибавить к каждому элементу один, и всё будет работать. Вот цикл, который выводит последовательность 1,2,3,1,2,3,1,2,3 с помощью этого метода.

```
$(5)(9)
$(3)(0)
{cycle}
$(4)(3$^3%^1+)
$(2)(4$)
$(3)(3$^1+)
#(3$^1-^5$-|)<cycle>
```

Вот такой вот код. Если же делать два вложенных цикла, то будет так:

```
$(3)(1)
{cycle1}
$(4)(1)
{cycle2}
$(2)(4$)
$(4)(4$^1+)
#(4$^1-^3-|)<cycle2>
$(3)(3$^1+)
#(3$^1-^3-|)<cycle1>
```

Как видите, код больше, хотя делает то же самое. Мы будем использовать первый вариант. Теперь давайте модифицируем этот цикл так, чтобы он выполнял нужное количество сравнений, исходя из размера массива. Количество сравнений подсчитывается по формуле $(N-1)^2$,

то есть 9 сравнений для 4-х элементного массива. Но мы должны отнять один, ведь начинаем считать с нуля, поэтому формула такая: $(N-1)^2 = N^2-2N+1 = N^2-N-N+1$. Как их много! Давайте посмотрим, какая из них самая короткая. Пусть N лежит в шестой ячейке.

$(N-1)^2$ = 2^6$^1-&

N^2-2N-1 = 2^6$&^6$^2*-^1+

$N^2-N-N+1$ = 2^6$&^6$-^6$-^1+

Как видите, первая самая короткая, мы её и будем использовать.

Итак, код такой:

```
$(5)(2^6$^1-&)
$(3)(0)
{cycle}
$(4)(3$^6$^1-%^1+)
$(2)(4$)
$(3)(3$^1+)
#(3$^1-^5$-|)<cycle>
```

Итак, с циклами мы вроде разобрались. Теперь давайте подумаем над перестановкой элементов, то есть, как менять их местами. Вначале надо сравнить их, какой из них больше, и по результату сравнения уже выяснить, делать переворот или нет. На блок - схеме показан условный переход, который происходит, если первый элемент не больше второго. Напишем этот переход.

```
#(10^4$+$^10^4$^1++$-!^1-|)<noex>
```

В этой команде меня смущает одна часть:

```
10^4$^1++
```

Она вычисляет адрес следующего элемента массива. Вроде, всё правильно, но её можно сократить:

```
11^4$+
```

Так немного короче. Запишем всю команду.

```
#(10^4$+$^11^4$+$-!^1-|)<noex>
```

Теперь о самом обмене элементами. Алгоритм такой: переместить во временную ячейку первый элемент, потом в первую (из обменивающихся) ячейку массива поместить вторую, а потом, во вторую – значение из временной ячейки (Рис 17h). Самый простой способ реализовать это – использовать в качестве временной ячейки ячейку памяти. Достаточно хороший способ, но у него есть один недостаток – он требует свободную ячейку. Для небольших программ это не проблема, а если же программа солидных размеров, то это не лучший метод.

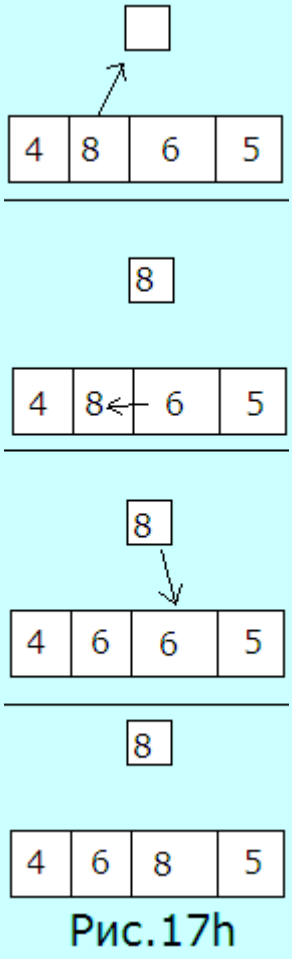


Рис.17h

Метод получше – использовать в качестве временной переменной стек. То есть берём в стек значение первой ячейки, перемещаем вторую в первую и во вторую переносим значение из стека. Получаем:

```
P(10^4$+$)
$(10^4$+)(11^4$+$)
$(11^4$+)(g)
```

Потом идёт метка, на которую переходит программа, если элементы стоят правильно.

```
{noex}
```

Наконец-то можно записать весь алгоритм в целостности.

```
$(5)(2^6$^1-&)
$(3)(0)
{cycle}
$(4)(3$^6$^1-%^1+)
#(10^4$+$^11^4$+$-!^1-|)<noex>
P(10^4$+$)
$(10^4$+)(11^4$+$)
$(11^4$+)(g)
{noex}
$(3)(3$^1+)
#(3$^1-^5$-|)<cycle>
```

Вот такой вот алгоритм. Чтобы он заработал, надо в шестую ячейку занести длину массива, а сам располагаться со смещением 10. А теперь, для закрепления знаний давайте напишем программку, которая просит ввести с клавиатуры массив, сортирует его и выводит.

Вначале надо написать участок кода, отвечающий за ввод массива. Мы его уже неоднократно писали, поэтому не будем на этом зацикливаться.

```
Program
$(3)(0)
{cycle1}
$(3)(3$^1+)
[Введите число. 0-выход]
$(4)(2$)
```

```
$(10^3$+)(4$)
#(4$|)<cycle1>
$(3)(3$^1-)
```

Далее, сортировка:

```
$(6)(3$)
$(5)(2^6$^1-&)
$(3)(0)
{cycle}
$(4)(3$^6$^1-%^1+)
#(10^4$+$^11^4$+$-!^1-|)<noex>
P(10^4$+$)
$(10^4$+)(11^4$+$)
$(11^4$+)(g)
{noex}
$(3)(3$^1+)
#(3$^1-^5$-|)<cycle>
```

И вывод массива:

```
[Отсортированный массив]
$(3)(1)
{cycle2}
$(2)(10^3$+$)
$(3)(3$^1+)
#(3$^1-^6$-|)<cycle2>
```

Запишем всё вместе, при этом немного оптимизируя.

```
Program
```

```
$(3)(0)
{cycle1}
$(3)(3$^1+)
[Введите число. 0-выход]
$(4)(2$)
$(10^3$+)(4$)
#(4$|)<cycle1>
$(6)(3$^1-)
$(5)(2^6$^1-&)
$(3)(0)
{cycle}
$(4)(3$^6$^1-%^1+)
#(10^4$+$^11^4$+$-!^1-|)<noex>
P(10^4$+$)
$(10^4$+)(11^4$+$)
$(11^4$+)(g)
{noex}
$(3)(3$^1+)
#(3$^1-^5$-|)<cycle>
[Отсортированный массив]
$(3)(1)
{cycle2}
$(2)(10^3$+$)
$(3)(3$^1+)
#(3$^1-^6$-|)<cycle2>
```

Этот код поддерживает только один тип сортировки – по возрастанию. А если нам надо, чтобы он сортировал по убыванию? Для этого в команде, которая определяет пропуск команды,нужно поменять условие перехода: если необходимо сортировать по возрастанию, то там надо поставить минус, а если по убыванию, то плюс.

По возрастанию:

```
#(10^4$+$^11^4$+$-!^1-|)<noex>
```

По убыванию:

```
#(10^4$+$^11^4$+$-!^1+|)<noex>
```

А сделайте так, чтобы программа вводила массив и спрашивала условие сортировки: по убыванию или по возрастанию, а потом сортировала и выводила.

4. Примеры программ

Вот мы и рассмотрели 2/3 книги, изучили все команды, алгоритмы и многое другое. Но у нас не хватает самого главного – опыта. А опыт, как известно, прямо пропорционален испорченному оборудованию. Нет, нет, не подумайте, что мы призываем ломать ваши компьютеры. Оборудованием в этом случае, являются ваши программы, а «испорченные» - значит не работающие, с ошибками. То есть, чтобы писать хорошо программы, надо учиться на своих ошибках, а чтобы были ошибки, надо писать программы, поскольку можно провести такую логическую цепочку: есть программы => есть ошибки => на ошибках учимся => нарабатываем опыт. Поэтому писать и еще раз писать программы.

Вообще-то, вы уже должны уметь писать программы, потому что мы изучили весь язык, алгоритмы, но опыта

комбинирования алгоритмов для написания хороших программ у вас не хватает. Поэтому сейчас мы рассмотрим несколько готовых программ, в процессе изучения которых вы не только закрепите полученные знания, но и лучше поймёте сам процесс использования алгоритмов.

4.1 Поиск простых чисел

Давайте напишем программу, которая производит поиск простых чисел, то есть чисел, которые делятся только на один и на себя. Давайте подумаем над тем, как это сделать. Это сделать просто – надо по очереди проверять каждое число, простое оно или нет. Если же оно простое, то выводим его на экран. А как же проверить, простое число или нет? Надо делить числа на все от двух до корня из числа, и, если хотя бы на одно из них делится, то, значит, число не простое, а если же не делится ни на одно, значит, оно простое, его надо вывести на экран.

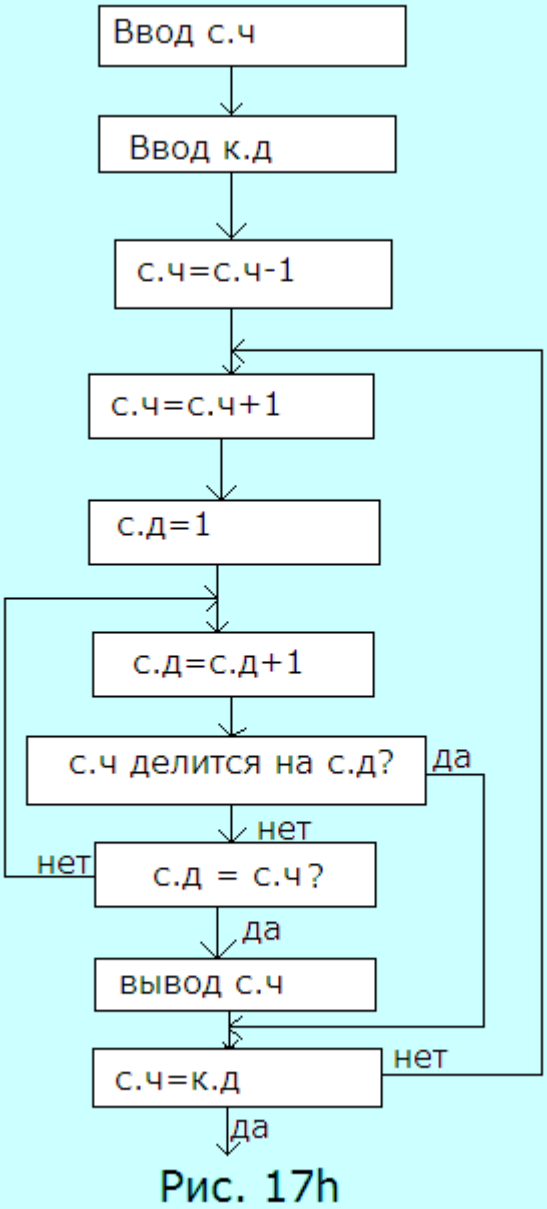


Рис. 17h

Давай теперь подумаем над алгоритмом. Он будет таким: два вложенных цикла, один считает от начала до конца периода, а второй от двух до числа, которое значится в первом цикле. Внутри же этого цикла происходит деление первого числа на второе, и, если оно делится, то прервать цикл и перейти к следующему числу. Вот блок-схема: (Рис.17h) Что

мы там видим? Вначале вводим значения диапазона поиска: с.ч – Счётчик чисел, т о есть, начало диапазона поиска, а потом к.д – конец диапазона.

Смотрим дальше. Уменьшаем с.ч. Зачем? Посмотрите, какая команда идёт следующей – с.ч=с.ч+1. То есть, мы вначале уменьшаем значение счётчика, так как в начале цикла он увеличивается. Таким образом, цикл начнёт работать с первым значением введённого диапазона.

Итак, цикл счёта чисел начался. Внутри его идёт цикл деления. Устанавливаем счётчик делителей на 1 и начинаем проверять. Увеличиваем счётчик, и получается 2 – первое число, на которое мы будем проверять делимость. Потом идёт самая важная команда – проверка, делится ли с.ч на с.д, и если оно делится, то производится выход из цикла. После этой команды идёт проверка, есть ли ещё числа, которые надо проверять, то есть, не равен ли счётчик делителей корню из счётчика чисел. Если же равен, то это значит, что число простое, так как за весь период не было найдено ни одного делителя, и это число надо вывести, что происходит в следующем действии. А если же не равно корню из с.ч, значит, диапазон ещё не кончился, и надо перейти на метку в начале цикла, считающего делители, где произойдёт очередная проверка на делимость.

И в самом конце происходит проверка, ни кончился ли диапазон проверяемых чисел, и если кончился, то программа завершается, иначе - цикл продолжается.

Да уж, трудновато, но что поделаешь, это же степлер. Давайте составим программу. Вот она:

```
Program
[Prime number Founder 2.0]
```

```
[From]
$(6)(2$^1-)
[To]
$(7)(2$)
{next}
$(6)(6$^1+)
$(4)(1)
{nl}
$(4)(4$^1+)
#(6$^4$%)<l>
#(4$^6$@-|)<nl>
$(2)(6$)
{l}
#(6$^7$-|)<next>
```

Эта программа достаточно хороша, считает довольно быстро, но есть один способ её ускорить. Обратите внимание но команду `#(4$^6$@-|)<nl>` В ней происходит постоянное вычисление корня, и происходит оно много раз с одинаковыми значениями, а это сильно замедляет выполнение программы. Поэтому сделаем так: в какую-то переменную положим результат вычисления корня и будем изменять значение этой переменной, когда меняется значение подкоренного числа. Таким образом в этой переменной будет постоянно находиться корень нужного числа.

Попробуем сделать это. Для начала надо взять какую-то свободную переменную. Пусть это будет пятая. Далее, надо оперативно изменять её, то есть, перемещать в неё

новое значение тогда, когда изменяется значение аргумента, то есть, шестой переменной, а это происходит только один раз – в начале цикла. Поэтому после этой команды поставим выражение $(5)(6@), а 6$@ заменим на 5$.

```
Program
[Prime number Founder 2.0]
[From]
$(6)(2$^1-)
[To]
$(7)(2$)
{next}
$(6)(6$^1+)
$(5)(6$@)
$(4)(1)
{nl}
$(4)(4$^1+)
#(6$^4$%)<l>
#(4$^5$-|)<nl>
$(2)(6$)
{l}
#(6$^7$-|)<next>
```

Вот какую «интересную» оптимизацию мы провели. Почему «интересную»? А потому, что размер программы не уменьшился, а, наоборот, увеличился. Это тоже может быть, поэтому такой метод надо иметь в виду.

Программа, которую мы сейчас написали – пример хорошей степлеровской программы. В ней мало чего лишнего, нет ненужных команд, всё оптимизированно. Но как побочный эффект – хорошие программы плохо читаемы, то есть, по исходнику достаточно трудно понять, что она делает. Но это не значит, что всякая запутанная программа – хорошая, всё должно быть в меру.

4.2 Пример работы с графикой

Делать программы, работающие с графикой достаточно интересно, потому что результат их работы может быть очень наглядный. Давайте попробуем что-то написать.

Процесс рисования может показаться вам очень примитивным и дедовским, но с его помощью можно достаточно быстро нарисовать хорошую картинку. Итак, берём тетрадь в клеточку и вырываем из неё двойной лист. Далее, в левом верхнем углу рисуем точку, чтобы она отступала от верха и слева одну клетку. Потом, от низа тоже до конца. Эти линии будут осями координат. . Нумеруем каждую линию вправо: 1,2,3,4,5… и вниз: 1,2,3,4,5,6…. (Рис.18h)

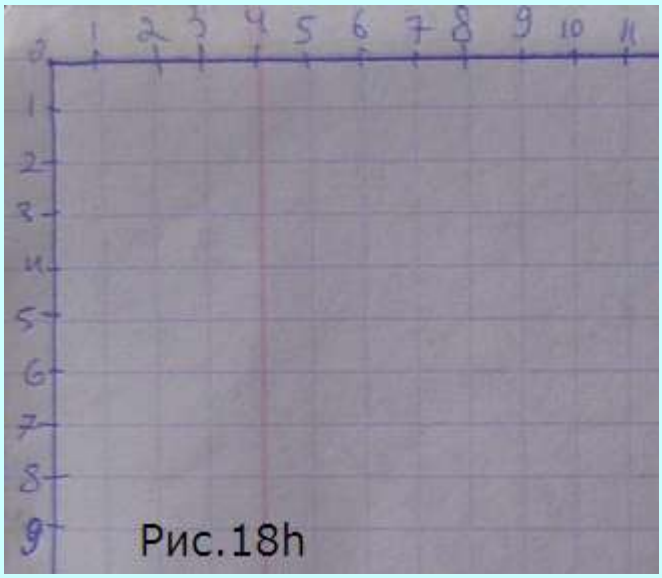


Рис.18h

Таким образом, мы получили проекцию экрана на бумаге, и теперь начинаем рисовать. Рисовать лучше карандашом, чтобы потом можно было стереть, и точки ставить в углах квадратиках. Как же теперь определять координаты точки? Провести мысленно линию к верху, пока не

коснётся нарисованной линии, и то число, которое будет возле точки соприкосновения, умноженное на 10, будет координатой по X. Теперь надо вычислить координату по Y. Делаем то же самое, только линию проводим не вверх, а влево. Давайте сделаем простой пример, напишем программу, которая выводит изображение (рисунок 19h) на экран.

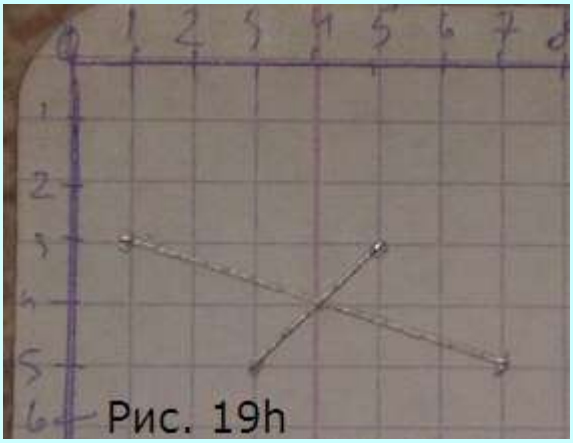


Рис. 19h

Берём первую линию. Как мы знаем, линия определяется двумя точками – началом и концом. Определим начало. Пусть это будет её левый конец. Проведём воображаемую линию вверх – она приходит к цифре 1, значит координата по X – будет 10. Проводим влево к цифре 3, значит, координата по Y будет 30. Это мы сделали первый конец, делаем второй. Проводим вверх – 7, влево – 5 , значит координаты конца – 70, 50. Линия пусть будет зелёным цветом, то есть, десятым, и тогда команда вывода линии принимает такой вид:

```
~(3)(10,30,70,50,10)
```

Теперь займёмся второй линией, у первого конца воображаемые линии приходят к цифрам 5 и 3, а второго конца – 3, 5 и пусть она будет жёлтого цвета. Команда будет выглядеть так:

```
~(3)(30,50,50,30,14)
```

Соберём всё вместе и получим такую программу:

```
Program
~(1)(1)
~(3)(10,30,70,50,10)
```

~(3)(30,50,50,30,14)

Вот мы и написали простейшую графическую программу.

Более сложные рисунки делаются также, только элементов там больше. Например:

~(1)(1)

~(3)(0,0,640,0,10)

~(3)(640,0,640,480,10)

~(3)(640,480,0,480,10)

~(3)(0,480,0,0,10)

~(3)(0,250,472,250,10)

~(3)(480,250,640,250,10)

~(3)(0,210,80,130,10)

~(3)(80,130,143,183,10)

~(3)(70,250,210,120,10)

~(3)(210,120,297,197,10)

~(3)(230,250,330,170,10)

~(3)(330,170,430,250,10)

~(3)(420,240,510,180,10)

~(3)(590,250,510,180,10)

~(3)(10,200,10,140,10)

~(3)(10,140,50,140,10)

~(3)(50,140,50,160,10)

~(3)(20,140,30,100,10)

~(3)(30,100,40,140,10)

~(4)(30,120,1,10)

~(3)(0,20,20,30,10)

~(3)(0,20,10,40,10)

~(3)(20,30,10,40,10)

~(3)(30,30,20,30,10)

~(3)(30,30,30,50,10)

~(3)(30,50,10,50,10)

~(3)(10,50,10,40,10)

~(3)(120,30,90,50,10)

~(3)(120,30,100,60,10)

~(3)(90,50,100,60,10)

~(3)(90,50,80,50,10)

~(3)(80,50,70,70,10)

~(3)(70,70,100,70,10)

~(3)(100,60,100,70,10)

~(3)(480,220,480,330,10)

~(3)(480,220,460,220,10)

~(3)(460,220,480,270,10)

~(3)(480,270,460,330,10)

~(3)(470,300,470,330,10)

~(3)(460,330,470,330,10)

~(3)(480,270,490,300,10)

~(3)(490,300,500,330,10)

~(3)(490,300,510,330,10)

~(3)(500,330,510,330,10)

~(3)(480,330,470,350,10)

~(3)(470,350,460,380,10)

~(3)(470,350,470,380,10)

~(3)(460,380,470,380,10)

~(3)(480,330,510,380,10)

~(3)(490,350,490,380,10)

~(3)(510,380,490,380,10)

~(3)(360,70,360,196,10)

~(3)(360,70,480,70,10)

~(3)(480,70,480,198,10)

~(4)(450,120,10,10)

~(4)(390,120,10,10)

~(3)(400,180,400,228,10)

~(3)(400,180,430,180,10)

~(3)(430,180,430,232,10)

~(3)(350,80,420,10,10)

~(3)(420,10,490,80,10)

~(17)(70,200,6,10)

~(17)(200,200,6,10)

~(17)(300,220,6,10)

~(17)(510,220,6,10)

~(17)(200,300,7,10)

~(17)(150,50,11,10)

~(17)(470,230,14,10)

~(17)(465,320,2,10)

~(17)(505,325,2,10)

```
~(17)(500,370,6,10)
~(17)(465,378,6,10)
~(17)(30,150,2,10)
~(17)(30,130,1,10)
~(17)(10,30,1,10)
~(17)(100,50,1,10)
~(17)(20,40,12,10)
~(17)(90,60,12,10)
~(17)(400,50,6,10)
~(17)(410,200,5,10)
~(17)(410,150,1,10)
~(17)(390,120,9,10)
~(17)(450,120,9,10)
```

Что рисует эта программа? А вот запустите, и сами увидите.

4.3 Шифровщик файлов

Иногда надо сделать так, чтобы файл никто кроме вас прочитать не смог. То есть, надо зашифровать его. Сейчас мы напишем программу, которая будет это делать.

Мы будем использовать XOR шифрование, то есть, делать каждому байту файла операцию XOR с другим байтом – некоторым ключом, который мы должны знать, потому что, только зная его, мы сможем прочитать файл. Вначале введем имя файла, который мы будем шифровать. Делаем это с помощью цикла, вводя по одной букве и оставляя эту букву в памяти. После ввода файла открываем его для чтения.

```
[Введите имя входного файла]
$(3)(10)
{inp}
$(3$)(1$)
$(1)(3$$)
$(3)(3$^1+)
#(3$^1-$^13-|)<inp>
$(3$^1-)(0)
$(1)(10)
~(10)(10,1,0)
```

Открываем ещё один файл, но теперь для записи, так как туда мы будем помещать зашифрованный файл.

```
[Введите имя выходного файла]
$(3)(10)
{inp2}
$(3$)(1$)
$(1)(3$$)
$(3)(3$^1+)
#(3$^1-$^13-|)<inp2>
$(3$^1-)(0)
$(1)(10)
~(10)(10,2,1)
```

Далее, когда мы открыли оба файла, можно начать шифровать, но прежде надо узнать ключ шифрования.

```
[Введите ключ шифрования. 0 - 255]
$(6)(2$)
```

Теперь, собственно, шифруем файл – читаем один байт, записываем его уже поксоренным, и так до тех пор, пока файл не кончится.

```
[Шифруем...]
{fil}
~(12)(1,5,11)
~(11)(2,5$^6$#)
#(11$|)<fil>
```

И закрываем файлы.

```
~(13)(1)
~(13)(2)
[Зашифровалось!!!]
```

Вот такая программа. Запишем её теперь полностью:

```
[Введите имя входного файла]
$(3)(10)
{inp}
$(3$)(1$)
$(1)(3$$)
$(3)(3$^1+)
#(3$^1-$^13-|)<inp>
$(3$^1-)(0)
$(1)(10)
~(10)(10,1,0)
[Введите имя выходного файла]
$(3)(10)
{inp2}
```

```
$(3$)(1$)
$(1)(3$$)
$(3)(3$^1+)
#(3$^1-$^13-|)<inp2>
$(3$^1-)(0)
$(1)(10)
~(10)(10,2,1)
[Введите ключ шифрования. 0 - 255]
$(6)(2$)
[Шифруем...]
{fil}
~(12)(1,5,11)
~(11)(2,5$^6$#)
#(11$|)<fil>
~(13)(1)
~(13)(2)
[Зашифровалось!!!]
```

Как же пользоваться этой программой? А вот так: запускаем её, вводим имя шифруемого файла, нажимаем энтер. Потом вводим имя файла, который получится при шифровании. Тоже нажимаем энтер. Затем вводим ключ шифрования – число от 0 до 255, так как этот ключ имеет размер один байт. И теперь начинается шифровка файла. Она занимает некоторое время.

Когда файл зашифровался, ищем тот файл, имя которого мы вводили вторым – это и будет зашифрованный файл. Но теперь возникает другой вопрос

– как расшифровать файл? А проще простого – надо снова зашифровать его, используя тот же ключ, что вы использовали при шифровании. И тогда выходным файлом будет как раз тот исходный файл. Вот так всё просто.

4.4 Поиск факториала

Поскольку степлер умеет вызывать процедуры и передавать параметры, значит, можно использовать рекурсию. Давайте попробуем написать программу, реализующую это.

Пусть это будет простейшая рекурсивная программа-поиск факториала. Начнём с того, что такое факториал. Факториал – это произведение чисел от одного до заданного числа. Например, факториал числа 5 (пишется 5!) – это 1*2*3*4*5=120.

Теперь о том, как его вычислять. Можно сделать просто цикл, который будет считать от одного до нужного числа, умножая их друг на друга, но можно сделать и по-другому алгоритму.

Этот алгоритм звучит так: чтобы найти факториал числа n, надо: Если n=1, то факториал n=1, а если n>1, то факториал n равен факториал n-1 умножить на n. Описание алгоритма в виде математической нотации дано на рисунке 1Ah. Как видите, такой тип описания алгоритма достаточно сильно отличается от блок - схемы, но он достаточно удобен, и его легко перевести в программу, что мы сейчас и сделаем.

$$fac(n)=\begin{cases} n=1 \mid fac(n)=1 \\ n>1 \mid fac(n)=fac(n-1)*n \end{cases}$$

Рис. 1Ah. Факториал

Для начала надо сделать процедуру, которая будет, собственно, вычислять факториал. Точнее, не процедуру, а её заголовок.

```
={fac}=
```

Далее, надо прочитать из стека параметров аргумент и занести во временную переменную, которую надо вначале объявить.

```
?[3]
$(3)(g)
```

Следом за этим надо проверить, не равен ли аргумент единице, и если равен, то перейти на метку.

```
#(3$^1-)<one>
```

Теперь, поскольку аргумент не равен нулю, можно вычислить выражение fac(n-1)*n. Вначале вычисляем факториал уменьшенного числа, а потом умножаем на n и помещаем это значение в стек.

```
*[3$^1-]<fac>
P(g^3$*)
```

Заканчиваем процедуру.

```
**
```

Вроде бы всё, но, кажется, мы что-то забыли. Да, метка one, на которую мы переходим, если аргумент равен нулю. Где она должна быть? Да, это достаточно интересно, но она должна стоять после завершения процедуры. Это странный метод, но он хорошо экономит место, когда у процедуры есть несколько точек выхода.

```
{one}
```

Помещаем в стек 1 и выходим.

```
P(1)
**
```

Вот и вся процедура. Запишем её полностью.

```
={fac}=
?[3]
$(3)(g)
#(3$^1-)<one>
*[3$^1-]<fac>
p(3$^g*)
**

{one}
p(1)
**
```

Как видим, эта процедура вызывает сама себя. Это называется рекурсией и достаточно часто применяется в математических вычислениях, например, при поиске наибольшего общего делителя или чисел фибоначчи.

Теперь надо дописать участок кода, который будет просить ввести число с клавиатуры, вычислять факториал и выводить его.

Это сделать просто.

```
[Введите число]
*[2$]<fac>
$(2)(g)
```

Запишем всё вместе:

```
={fac}=
?[3]
$(3)(g)
#(3$^1-)<one>
*[3$^1-]<fac>
p(3$^g*)
**
{one}
p(1)
**
Program
[Введите число]
*[2$]<fac>
$(2)(g)
```

Попробуйте написать эту же программу, только чтобы она вычисляла факториал не рекурсивным, а обычным циклическим методом.

4.5 Программа, угадывающая числа

Давайте сделаем такую программу, которая угадывает задуманное вами число. То есть, вы загадываете число от одного до ста, а компьютер угадывает его, задавая вам вопросы типа: «это число больше 50?»

Вначале надо продумать алгоритм, по которому будет происходить угадывание. Вы, наверное, удивитесь, но на самом деле вы уже знаете, как это сделать. Не знаете? Тогда скажите, изучали ли мы поиск в отсортированном массиве? Изучали. Так вот, мы будем делать эту

программу, используя тот алгоритм. Так что же общего между этими двумя программами? А вот что: числовая прямая и отсортированный массив чем-то друг на друга похожи. А это значит, что и способы их поиска должны быть похожими.

Как мы будем искать? У нас есть область, в которой находится загаданный элемент. Мы берём число из середины этой области и спрашиваем пользователя: больше, меньше или равно оно загаданному? Пользователь отвечает на этот вопрос, и, опираясь на его ответ, мы делаем соответствующие действия с границей: усекаем её слева или справа.

А вот и сама программа:

```
Program
$(5)(1)
$(6)(100)
{cycle}
$(3)(5$^6$+^2/)
#(5$^6$-)<ugadal>
[Это число]
$(2)(3$)
[0 - Равно, 1 – Надо меньше, 2 – Надо больше]
$(4)(2$)
#(4$)<ugadal>
#(4$^1-)<menshe>
$(5)(3$^1+)
#<cycle>
```

```
{menshe}
$(6)(3$^1-)
#<cycle>
{ugadal}
[Ура!]
```

Как видим, отличий от программы поиска в массиве достаточно мало. Поэтому подробно изучать программу не будем.

Сейчас мы познакомились с одним интересным свойством: алгоритмы, созданные для одних целей, могут использоваться в совсем других целях, и такое бывает довольно часто.

4.6 Пример работы с дефинами

До этого момента мы изучали, как надо программировать, а вот программа, которая показывает, как не надо программировать.

Она очень перегружена дефинами, из неё совершенно непонятно, что она делает. Вот, посмотрите сами. Вначале идёт куча определений дефин, а потом сонет Шекспира. Так вот, этот сонет и представляет собой программу.

```
Define Уильям_Шекспир=''
Define Сонет_14=''
Define Пусть='='
Define лишь='{'
Define отчасти='fac'
Define мне='}'
Define знаком='='
Define язык=''

Define Небесных='?'
```

```
Define звезд='[3'
Define я='4'
Define тоже=']'
Define астроном=''

Define Хоть_я='$(3)'
Define судить='('
Define по='G'
Define звездам=')'
Define не_привык=''

Define О_потрясеньях='#(3$^1-)'
Define на='<'
Define пути='one'
Define земном='>'

Define Не_знаю=''
Define как='*[3$^1-]'
Define предречь='<'
Define минутам='fac'
Define срок='>'

Define И_дождь='P'
Define благоприятный='(3$'
Define для='^G'
Define полей='*)'

Define Читать='*'
Define я_не_умею='*'
Define звездных=''
Define строк=''

Define Не_смею='{'
Define обнадежить='one'
Define королей='}'

Define НО_мне='P'
Define читать_в='('
```

```
Define твоих='1'
Define глазах=')'
Define дано=''

Define В_надежных='*'
Define звездах='*'
Define даже=''
Define в_наши_дни=''

Define Что='*'
Define красота='[2$]'
Define и_правда='<fac>'
Define заодно=''

Define и_лишь_в='$(2)'
Define твоих_глазах='(G'
Define живут=')'
Define они=''

Define глаза_твои=''
Define открыли=''
Define мне_секрет=''

Define Нет_красоты=''
Define без=''
Define них=''
Define и_правды_нет=''

Уильям_Шекспир
Сонет_14

Пусть лишь отчасти мне знаком язык
Небесных звезд, я тоже астроном
Хоть_я судить по звездам не_привык
О_потрясеньях на пути земном;
Не_знаю как предречь минутам срок
И_дождь благоприятный для полей;
Читать я_не_умею звездных строк
```

Не_смею обнадежить королей;
Но_мне читать_в твоих глазах дано
В_надежных звездах даже в_наши_дни
Program
Что красота и_правда заодно
И_лишь_в твоих_глазах живут они;
Глаза_твои открыли мне_секрет
Нет_красоты без них и_правды_нет

А действительно, что делает эта программа? А вот что – она вычисляет значение факториала числа, то есть, если мы уберём все определения, то получится уже знакомая нам программа:

```
={fac}=
?[3]
$(3)(g)
#(3$^1-)<one>
*[3$^1-]<fac>
p(3$^g*)
**
{one}
p(1)
**
Program
*[2$]<fac>
```

Да, кстати, забыл сказать, эту программу написал Абадябер.

ОЙ-ОЙ-ОЙ!

Эта программа почему-то не работает в интерпретаторе TOTOP. Но надеюсь, что в

будущем исправлю эту ошибку.

4.7 Пишем библиотеку

Если на степлере писать серьёзные программы, то может показаться, что в нём не хватает некоторых очень важных команд, например, команд работы со строками. Но это не беда, в степлере можно создавать библиотеки, и об одной из них мы вам поведаем.

Библиотека, которую мы будем рассматривать, была написана Абадябером, и содержит процедуры разного назначения. Вся информация о процедурах, передаваемых параметрах и прочем. указана в исходнике, поэтому повторяться не буду. Вот её код:

```
;Utils.suf. Утилиты самого разного назначения

;Процедура ReadLn(<VarIndex>).
;Ждет ввода строки пользователем. Строка помещается начиная
;с переменной <VarIndex>
;Поддержка BackSpace. После ввода строки переносит строку.
={ReadLn}=
   ?[3,4]
   $(3)(G)
   $(4)(3$)
 {Inp}
   $(3$)(1$)
   #(3$$)<InpExt> ;Расширенные скан-коды не обрабатывать
   $(1)(3$$)
   $(3)(3$^1+)
   #(3$^1-$^8-)<BackSp>
   #(3$^1-$^13-|)<Inp>
   #<EndInp>
 {BackSp}
```

```
   #(4$^3$-)<Inp>
   $(3)(3$^2-)
   $(1)(32)
   $(1)(8)
   #<Inp>
 {InpExt}
   $(4)(1$)
   #<Inp>
 {EndInp}
   $(3$^1-)(0)
   $(1)(10)
**

;Процедура WriteLn(<VarIndex>)
;Печатает на экране текстовую строку, начинающуюся
;с VarIndex, до первого нуля. После печати ставит
;символы CR и LF.
={WriteLn}=
   ?[3]
   $(3)(G)
 {WriteNextChar}
   #(3$$)<EOStr>
   $(1)(3$$)
   $(3)(3$^1+)
   #<WriteNextChar>
 {EOStr}
   $(1)(13)
   $(1)(10)
**

;Процедура Write(<VarIndex>)
;Печатает на экране текстовую строку, начинающуюся
;с VarIndex, до первого нуля.
={Write}=
   ?[3]
   $(3)(G)
 {WriteNextChar2}
   #(3$$)<EOStr2>
```

```
   $(1)(3$$)
   $(3)(3$^1+)
   #<WriteNextChar2>
  {EOStr2}
**

;Функция FReadLn(<FileIndex>, <VarIndex>): <IsEOF>
;Читает из файла <FileIndex> текстовую строку, и
;помещает ее, начиная с переменной <VarIndex>
;IsEOF будет 0, если конец файла был достигнут, и 1
;в противном случае. Файл должен быть открыт.
={FReadLn}=
   ?[3,4,5]
   $(3)(G)
   $(4)(G)
  {ReadNextByte}
   ~(12)(4$,3$,5)
   $(3)(3$^1+)
   #(5$)<EOL> ;Проверка на конец файла
   #(3$$^10-)<EOL> ;Проверка на конец строки
   #(3$$^13-)<EOL>
   #<ReadNextByte>
  {EOL}
   $(3$)(0)
   P(5$)
**

;Процедура FWriteLn(<FileIndex>, <VarIndex>)
;Записывает в файл <FileIndex> текстовую строку,
;с переменной <VarIndex>. Файл должен быть открыт.
={FWriteLn}=
   ?[3,4]
   $(3)(G)
   $(4)(G)
  {WriteNextByte}
   ~(11)(4$,3$)
   $(3)(3$^1+)
   #(3$$)<EOL2> ;Проверка на конец строки
```

```
  #<WriteNextByte>
 {EOL2}
**
```

Как использовать эту библиотеку? Да очень просто: Скопируйте этот текст в файл utils.suf, поместите его в папку с программой, и в программе вначале напишите uses ‘utils.suf’. Теперь можете вызывать процедуры из библиотеки, как будто они лежат в вашей программе.

4.8 Работа с мышью

Вот пример достаточно серьёзной программы на степлере Она иллюстрирует работу с мышью через системные прерывания. Работает она так: на графическом экране нарисована стрелка мыши, и её можно двигать. За мышкой остаётся след из разноцветных точек.

Если же нажать кнопку мыши, то экран заполнится разноцветными полосками.

Автор этой программы – Абадябер.

```
;mouse.st. Запилил Абадябер =)
;Является примером, иллюстрирующим активную
работу с подпрограммами, определениями define,
;и специальными функциями языка, такими, как
работа с портами и вызов прерываний BIOS.
;Вообще, все это дело вполне можно запилить в
библиотеку, ибо фактически это она самая и есть ;)

Define Regs='200' ;Регистры будут расположены с 200
переменной
Define MouseInfo='10'
Define Screen13='19' ;Экранный режим. 19 -
320x200x256;  18 - 640x480x16  и.т.п
Define TextMode='3'  ;Текстовой режим.
Define X='11'
Define Y='12'
Define C='13'
```

```
={InitMouse}=  ;Инициализирует мышь
   ?[3]
   $(3)(G)
   $(3$)(0) ;AH = 0
   $(3$^1+)(0) ;AL = 0
   ~(9)(51,3$)
   P(3$$)
**

={ShowMouse}=   ;Показывает курсор мыши
   ?[3]
   $(3)(G)
   $(3$)(0)
   $(3$^1+)(1) ;Al = 1
   ~(9)(51,3$)
**

={HideMouse}=   ;Прячет курсор мыши
   ?[3]
   $(3)(G)
   $(3$)(0)
   $(3$^1+)(2) ;Al = 2
   ~(9)(51,3$)
**

={CheckMouse}=    ;Получает информацию о
состоянии мыши
   ?[3]
   $(3)(G)
   $(3$)(0)
   $(3$^1+)(3) ;Al = 1
   ~(9)(51,3$)
   P(3$^4+$^256*^3$^5+$+)    ;Push cx
   P(3$^6+$^256*^3$^7+$+)    ;Push dx
   P(3$^2+$^256*^3$^3+$+)    ;Push bx
**

={Scr}=   ;Устанавливает экранный режим
```

```
   ?[3,4]
   $(4)(G)
   $(3)(G)
   $(3$)(0)  ;AH = 0
   $(3$^1+)(4$)  ;AL = 4$
   ~(9)(16,3$)
**

={PSet}=  ;Устанавливает точку по переданным
координатам и цвету.
   ?[3,4,5,6]
   $(6)(G)
   $(5)(G)
   $(4)(G)
   $(3)(G)
   $(3$)(12)
   $(3$^1+)(6$)
   $(3$^2+)(0)
   $(3$^7+)(4$^256%)
   $(3$^6+)(4$^256/)
   $(3$^5+)(5$^256%)
   $(3$^4+)(5$^256/)
   ~(9)(16,3$)
**

Program
*[Regs,Screen13]<Scr>
*[Regs]<InitMouse>
$(MouseInfo)(G)
*[Regs]<ShowMouse>
$(X)(0)
$(Y)(0)
$(C)(0)
{Loop}
   *[Regs]<CheckMouse>
   $(C)(G)
   $(Y)(G)
   $(X)(G)
```

```
    #(C$|)<Next>
    *[Regs,Y$,X$^2/,X$^16%]<PSet>
#<Loop>
{Next}
$(X)(0)
$(Y)(0)
$(C)(0)
{Draw}
    *[Regs,Y$,X$,C$]<PSet>
    $(X)(X$^1+)
    #(X$^319-|)<Draw>
    $(X)(0)
    $(Y)(Y$^1+)
    $(C)(C$^1+)
#(Y$^199-|)<Draw>
*[Regs,TextMode]<Scr>
```

5. Интерпретаторы

А теперь давайте подумаем, как же всё-таки запустить все те программы, которые мы тут с вами написали. Это можно сделать с помощью интерпретаторов, и мы представляем вам целых два.

5.1 Тотор 3.1

Интерпретатор Тотор – это мой интерпретатор, то есть, я являюсь его автором. Сейчас мы рассмотрим его плюсы и минусы.

Из плюсов следует отметить следующее: Тотор содержит встроенный редактор программ, поэтому сразу, как только вы его установили, можно приступать к работе – писать программы, запускать, тестировать. Надо отметить, что этот редактор написал не я, а Сергей Чехута, поэтому все претензии и пожелания отправляйте ему. Также из плюсов стоит отметить достаточно краткий

исходный код интерпретатора. Он написан на языке Borland Pascal и занимает около 750 строк. Для кого-то это покажется много, но для интерпретатора языка это не много.

А из минусов же стоит отметить скорость интерпретации, которая ниже, чем у интерпретатора Lint, но всё -таки находящуюся на вполне приемлемом уровне. Также можно отметить отсутствие компилятора и другие мелочи, которые я постараюсь исправить в ближайшем будущем.

А что же означает слово Тотор? Нет, это не аббревиатура, и не пытайтесь его расшифровать, Тотор – это животное, созданное мной в дурацкой игре SPORE. А почему я интерпретатор назвал в честь него? Да просто это было первое, что пришло на ум, вот и всё.

Скачать интерпретатор можно здесь:

Kabardcomp.narod.ru/totop3.rar

5.2 Lint 3.0.3

Есть ещё один интерпретатор степлера – Lint. Он написан Абадябером и является очень прогрессивной программой. Рассмотрим его особенности.

Вначале достоинства: интерпретатор Lint, помимо интерпретации, может создавать исполняемые файлы, и при этом, двумя способами: первый - создаётся экзешник, который содержит в себе и сам интерпретатор, и исполняемую программу, и можно запускать программу, не имея при себе интерпретатора. Второй способ создания EXE файлов – компиляция. Тут команды степлера переводятся сразу в ассемблерный код, который можно

скомпилировать ассемблером FASM. Достоинство этого метода в том, что получаются очень маленькие и быстрые программы, а недостаток в том, что не каждую программу можно откомпилировать.

Кроме компиляции к достоинствам этого интерпретатора можно отнести и высокую скорость работы – на порядок большую, чем в Тоторе. Это достигается благодаря тому, что программа, перед тем, как выполнятся, преобразуется в некий промежуточный байт-код, который выполняется намного быстрее.

Теперь о недостатках, которых у Lint'а совсем не много. Во-первых, он не имеет IDE. Но это легко исправляется, ведь можно прикрутить туда любое IDE, хотя бы тот же Turbo Shell из комплекта Тотор. Ещё одним недостатком можно считать более запутанный и сложный исходный код, хотя на программировании степлер программ это никак не сказывается.

Логотипом этого интерпретатора является Rainbow Dash – герой из одного очень милого, но не менее зомбирующего мультсериала.

Скачать его (интерпретатор) можно здесь: www.abaduaber.narod.ru/my/lastlint.zip

5.3 Изменение кодировок

Все программы, написанные в этой книге, представлены в кодировке Win-1251, или, проще говоря, в

Windows кодировке. Однако, существующие на данный момент интерпретаторы понимают только кодировку CP-886, то есть DOS – кодировку. Так как же перевести из одной кодировки в другую?

Вначале нужно сказать, что если программа не содержит русских букв, то можно кодировку не менять, так как коды этих символов в этих кодировках совпадают. Но если же есть русские буквы, то перекодировка нужна.

С помощью какой программы это можно сделать? Ответ: это можно сделать программой SNK Text Decoder, которую можно скачать здесь: Kabardcomp.narod.ru/decode.rar

Эта программа имеет такой вид: (Рис.18h)

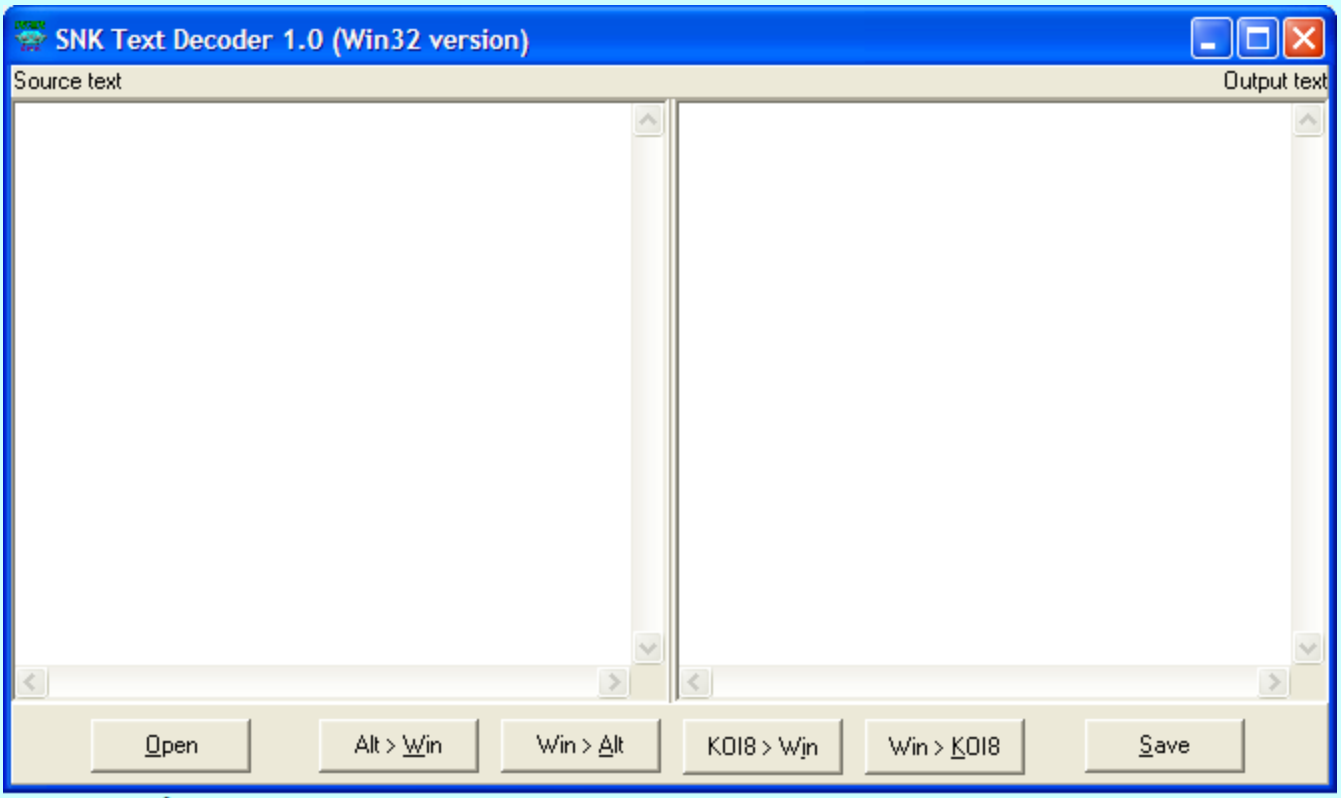


Рис.18h Декодер

Как же с ней работать? А вот так: нажимаем Open и открываем файл в кодировке Windows. Далее, нажимаем кнопку Win>Alt (Dos кодировка здесь названа Alt), и в

правом поле появляется этот же текст, но в досовской кодировке. Нажимаем Save и сохраняем файл. Вот и всё.

6. Заключение

Вот и подошло к концу наше изучение степлера. Надеюсь, что вам было интересно, и вы хорошо изучили язык, и у вас не разорвало мозг от изобилия информации. Желаем вам писать на степлере хорошие, интересные программы, которые будут делать много интересных вещей. Если же не получается, то не расстраивайтесь, просто перечитайте нужные главы, и всё должно получиться.

Так же стоит отметить, что язык сейчас бурно развивается, и, возможно, уже сейчас есть версия языка, которая более новая, чем описываемая в книге. Так что удачи вам в этом интересном деле.

Но подождите, мы тут всё про степлер, да про степлер, и как-то опускаем этот вопрос: а кто же такой Kakos_nonos? Думаю, ответ на этот вопрос будет более длинный, чем сама книга, поэтому скажу вкратце: Kakos_nonos – очень популярный в последнее время интернет мем. Был придуман Kakos_nonos'ом, то есть самим собой, что противоречит закону сохранения массы, но ничего не поделаешь, что было – то было. Первое упоминание о нём было где-то в 2009 году, и с тех пор он неоднократно терроризирует интернеты в целях, неизвестных никому, кроме как ему самому.

Однако, в последнее время стали плодиться слухи, что Kakos_nonos вовсе не мем, а реальное физическое лицо, обитающее на земле. Хотя эти слухи можно считать недействительными, так как ни одной фотографии, и тем более, видеозаписи с ним нет и не будет. Вот так.

Приложение 1. Описание языка

Язык программирования степлер. Версия 3.1

Описание языка.

1.Структура программы.

<Раздел define>

<Раздел uses>

<Подпрограммы>

Program

<Программа>

1.1 Раздел define

define <идентификатор> = '<Замена>'

define <идентификатор> = '<Замена>'

define <идентификатор> = '<Замена>'

...

Эта команда заменяет один текст в программе на другой.

Она действует только на тот текст, который находится после команды define.

Команду define можно писать в любом месте программы

Идентификатор должен начинаться с латинской буквы.

Максимальная длина идентификатора и замены - 20 символов

Пример: define initgraph = '~(1)(1)'

команда define регистрозависима

1.2 Раздел uses

uses '<имя>'

uses '<имя>'

uses '<имя>'

...

<имя> = <имя_файла>.suf (Stepler Uses File)

Эта команда вставляет содержимое другого файла в основную программу в то место, где находится

uses.

uses можно писать в любрм месте программы.

Пример: uses 'sound.suf'

команда uses регистрозависима

1.2.1 Структура .suf файла

<Раздел define>
<Раздел uses>
<Подпрограммы>
<Программа>

1.3 Подпрограммы

=<имя подпрограммы>=

<Объявление локальных переменных>

<Подпрограмма>

**

Имя подпрограммы не регистрозависимо, то есть ={Proc}= и ={pRoC}= это одно и тоже.

Также можно называть процедуры русскими именами.

1.3.1 Объявление локальных переменных

?[<номер переменной>,<номер переменной>,<номер переменной>, ...]

Здесь указываются номера переменных, которые будут использоваться на нужды подпрограммы.

Подпрограмма может менять значения этих переменных, но основная программа этого не заметит, после

выполнения подпрограммы эти переменные будут иметь теже значения, что и до подпрограммы.

Пример: ?[5,3,7,3]

1.4 Программа

Программа начинается после слова Program.

Состоит из списка команд, где на одной строке может быть только одна команда.

<команда>

<команда>

<команда>

<команда>

...

2 Команды

В степлере есть несколько основных команд:

Команда присваиваия

$(<выражение>)(<выражение>)

Команда перехода на метку

#(<выражение>)< <метка> >

Команда вывода строки

[<строка>]

Команда вызова подпрограммы

*(<выражение>)[<параметр>,<параметр>,<параметр>, ...]< <имя_подпрограммы> >

Команда вызова специальной функции

~(<выражение>)(<параметр>,<параметр>,<параметр>, ...)

Команда записи в стек параметров

P(<выражение>)

Прямая пересылка памяти.

M(<сегмент источник>,<смещение источник>,<сегмент приёмник>,<смещение приёмник>,<кол-во байт>)

Комментарий до конца строки

/<комментарий>

2.1 Команда присваивания

Команда имеет такой синтаксис:

$(<выражение>)(<выражение>)

Выражение в первых скобках означает номер переменной, в которую записывать число, а вторая - что записывать.

Пример: $(5)(7) - Записать в пятую ячейку число семь.

Если же во время вычисления выражения в стеке остались значения, то первое значение (не вершина)

копируется в указанную переменную, а другие значения в следующие.

Пример: $(5)(5^4) - В пятую переменную - 5, в шестую - 4.

2.1.1 Переменные

степлере для хранения данных используются пронумерованные переменные. Все они имеют тип знаковое 16-битное целое.

Первые две зарезервированы для ввода/вывода.

Первыая - для символьного ввода/вывода.

Вторая - для численного.

2.3 Команда перехода на метку

Есть два типа перехода на метку - условный и безусловный.

Безусловный:

#< <метка> >

Условный:

#(<выражение>)< <метка> >

В условном переходе переход происходит только тогда, когда выражение в скобках равно нулю.

Метка определяется так: {<метка>}

Имена меток не регистрозависимы, и могут состоять из русских и английских букв.

Пример: #(0)<label>

2.4 Вывод строки

Выводится тот текст, который находится в квадратный скобках.

[<Выводимый текст>]

2.5 Команда вызова подпрограммы

Есть 4 типа вызова подпрограммы

Безусловный вызов без передачи параметров

*< <метка> >

Условный вызов без передачи параметров

*(<выражение>)< <метка> >

Безусловный вызов с передачей параметров

*[<параметр>,<параметр>,<параметр>, ...]< <метка> >

Условный вызов с передачей параметров

*(<выражение>)[<параметр>,<параметр>,<параметр>, ...]< <метка> >

В условном вызове вызов происходит только тогда, когда выражение равно нулю.

Параметры, передающиеся процедуре заносятся в стек параметров в направлении слево-направо.

Пример: *(0)[3,6,2,12]<proc>

2.6 Команда вызова специальнях функций

Эта команда имеет слудующий формат:

~(<выражение>)(<параметр>,<параметр>,<параметр>, ...)

В первой скобке идёт номер функции. Во второй идут параметры.

Есть 20 функций:

~(1)(режим) - изменить графический режим. Есть 2 режима:

0-текстовой

1 - 640*480*256

~(2)(X,Y,цвет) - рисовать точку в указанном месте с указанным цветом

~(3)(X,Y,X1,Y1,цвет) - рисовать линию

~(4)(X,Y,R,цвет) - рисовать круг

~(5)(X,Y,н.п) - получить цвет пикселя. Этот цвет записывается в ячейку н.п

~(6)(частота) - играть звук с частотой "частота".

~(7)(номер порта,значение) - отправить "значение" в порт "номер порта"

~(8)(номер порта,номер переменной) - прочитать значение из порта "номер порта" и отправить значение в

переменную "номер переменной"

~(9)(номер прерывания,смещение переменной) - выполнить прерывание МС-ДОС. Выполняется

прерывание "номер прерывания". Регистры, которые используются находятся со смещением "смещение

переменной". Регистры расположены в таком порядке: ah,al,bh,bl,ch,cl,dh,dl,es,ds,si,di,bp. Тоесть,если смещение

указано в 6, то ah будет в шестой переменной, al в седьмой, и. т. д.

~(10)(смещение к имени файла,номер файла,подфункция) - открыть фаил. Открывается фаил, название

которого находится в памяти со смещением "смещение к имени файла". Имя файла должно заканчиваться нулевым

символом.

Подфункция 0 - открыть для чтения.

Подфункция 1 - для записи.

Подфункция 2 - Переместится в файле в позицию смещение.

~(11)(Номер файла,байт) - Записать байт в фаил "номер файла".

~(12)(Номер файла,байт,конец) - прочитать байт из файла. Если фаил кончился, то переменная "конец"

примет значение в 0, иначе в 1.

~(13)(Номер Файла) - Закрыть фаил.

~(14)(номер цвета,р,г,б) - установить цвет "номер цвета" в р.г.б

~(15)(пауза) - пауза в сотых долях секунды.

~(16)(0) - очистить экран

~(17)(X,Y,C,G) - заливка. X,Y - точка заливки, C - цвет, G - цвет границы.

~(18)(цвет,размер,X,Y,смещение к началу текста) вывод текста в графическом режиме. Конец строки

обозначается нулевым символом.

~(19)(цвет,размер,X,Y,число) - вывести на экран число.

~(20)(цвет,размер,X,Y,код символа) вывод символа на экран.

Пример: ~(3)(4,2,64,22,15)

2.7 Команда записи в стек параметров

P(<выражение>,<выражение>,<выражение>, ...)

Эта команда записывает данные в стек параметров.

2.8 Комментарии

Комментарий до конца строки: /<комментарий>

2.9 Прямая пересылка памяти

M(<сегмент источник>,<смещение источник>,<сегмент приёмник>,<смещение приёмник>,<кол-во байт>)

3 Запись выражений

В степлере для записи выражений используется обратная польская нотация.

<выражение> = <оператор><оператор><оператор>...<оператор>

3.1 Числа

1, 10, 552, 704 - десятичная система счисления

~HA, ~H7B, ~H86C - шестнадцатеричная

~B101, ~B11001010 -двоичная

3.2 Операторы

Все операторы берут значения из стека, и возвращая результат туда же.

Операторы удаляют из стека те данные, которые берут.

+ - сложить вершину и подвершину

- - отнять от подвершины вершину

* - умножить вершину на подвершину

/ - делить вершину на подвершину

% - получить остаток от деления вершины на подвершины

^ - сдвинуть все значения стека на одив вверх

! - получить знак числа. >0 => 1; <0 => -1; 0 =>0

| - отрицание на 0. 0 => 1; <>0 =>0

@ - получить квадратный корень

$ - получить значение переменной, которое на вершине стека

G - получить число из стека параметров

? - случайное число от нуля до числа

" - из вершины стека загружается код символа, а потом выполняется оператор, соответствующий этому

символу.

. – Логическое И над вершиной и подвершиной.

\ - Логическое ИЛИ.

: - Отрицание вершины.

- Исключающие или.

' - переключить интерпретатор в режим, когда в стек помещается ASCII-код символа. Перейти в

нормальный режим можно с помошью этого же оператора.

S - указывает сегмент памяти, в котором находятся переменные.

Пример: 4$^5^4$%$+G^-

Приложение 2. ASCII – таблица

(Таблица на следующей странице)

866 MS-DOS Cyrillic CIS 1

	20	30	40	50	60	70	80	90	A0	B0	C0	D0	E0	F0
0	32	0 48	@ 64	P 80	` 96	p 112	А 128	Р 144	а 160	▓ 176	└ 192	╨ 208	р 224	Ё 240
1	! 33	1 49	A 65	Q 81	a 97	q 113	Б 129	С 145	б 161	▒ 177	┴ 193	╤ 209	с 225	ё 241
2	" 34	2 50	B 66	R 82	b 98	r 114	В 130	Т 146	в 162	■ 178	┬ 194	╥ 210	т 226	Є 242
3	# 35	3 51	C 67	S 83	c 99	s 115	Г 131	У 147	г 163	│ 179	├ 195	╙ 211	у 227	є 243
4	$ 36	4 52	D 68	T 84	d 100	t 116	Д 132	Ф 148	д 164	┤ 180	─ 196	╘ 212	ф 228	Ї 244
5	% 37	5 53	E 69	U 85	e 101	u 117	Е 133	Х 149	е 165	╡ 181	┼ 197	╒ 213	х 229	ї 245
6	& 38	6 54	F 70	V 86	f 102	v 118	Ж 134	Ц 150	ж 166	╢ 182	╞ 198	╓ 214	ц 230	Ў 246
7	' 39	7 55	G 71	W 87	g 103	w 119	З 135	Ч 151	з 167	╖ 183	╟ 199	╫ 215	ч 231	ў 247
8	(40	8 56	H 72	X 88	h 104	x 120	И 136	Ш 152	и 168	╕ 184	╚ 200	╪ 216	ш 232	° 248
9	) 41	9 57	I 73	Y 89	i 105	y 121	Й 137	Щ 153	й 169	╣ 185	╔ 201	┘ 217	щ 233	∙ 249
A	* 42	: 58	J 74	Z 90	j 106	z 122	К 138	Ъ 154	к 170	║ 186	╩ 202	┌ 218	ъ 234	· 250
B	+ 43	; 59	K 75	[91	k 107	{ 123	Л 139	Ы 155	л 171	╗ 187	╦ 203	█ 219	ы 235	√ 251
C	, 44	< 60	L 76	\ 92	l 108	| 124	М 140	Ь 156	м 172	╝ 188	╠ 204	▬ 220	ь 236	№ 252
D	- 45	= 61	M 77	] 93	m 109	} 125	Н 141	Э 157	н 173	╜ 189	═ 205	▌ 221	э 237	¤ 253
E	. 46	> 62	N 78	^ 94	n 110	~ 126	О 142	Ю 158	о 174	╛ 190	╬ 206	▐ 222	ю 238	■ 254
F	/ 47	? 63	O 79	_ 95	o 111	⌂ 127	П 143	Я 159	п 175	┐ 191	╧ 207	▀ 223	я 239	NBSP 255

Приложение 3. Таблица цветов

0 – Чёрный

1 – Тёмно синий

2 – Тёмно зелёный

3 – Серо - синий

4 - Коричневый

5 – Тёмно - сиреневый

6 – Оливковый

7 – Светло серый

8 – Тёмно серый

9 - Синий

10 - Зелёный

11 - Голубой

12 - Красный

13 - Розовый

14 - Жёлтый

15 – Белый

Конец