

Билет 1

1. Формализация алгоритма (стр.39)

Алгоритм - точно заданная последовательность правил, указывающая, каким образом можно за конечное число шагов получить выходное сообщение определенного вида, используя заданное входное сообщение. При этом подчеркивается, что действия всем понятны и легко выполнимы. Формализация алгоритма реализуется с помощью построения алгоритмических моделей. Можно выделить три основных типа универсальных алгоритмических моделей:

- рекурсивные функции (понятия алгоритма связывается с вычислениями и числовыми функциями),
- машины Тьюринга (алгоритм представляется как описание процесса работы некоторой машины, способной выполнять лишь небольшое число весьма простых операций),
- нормальные алгоритмы Маркова (алгоритмы описываются как преобразования слов в произвольных алфавитах).

Этими моделями мы формализуем понятие алгоритма.

Использование алгоритмических моделей опирается на тезис Тьюринга-Чёрча о том, что всякий интуитивный алгоритм может быть выражен средствами одной из алгоритмических моделей. Та же доказана, что всякий алгоритм, описанный средствами одной из моделей, может быть описан средствами другой. Проще говоря, одни модели сводятся к другим.

2. Блочная структура(стр.182)

Блок является простейшей формой программной единицы, представляющей собой обособленный фрагмент программы со своим локальным контекстом.

Определим блок как разновидность составной инструкции языка программирования, обозначаемую открывающей скобкой (`begin`, `{`) и закрывающей скобкой (`end`, `}`) и имеющую следующую структуру:

`begin // Начало блока языка АЛГОЛ–60`

`<последовательность описаний>\`

`<последовательность инструкций>`

`end // Конец блока`

Как и составной оператор, блок может быть вложенным и входить в другой, охватывающий блок. Во вложенном блоке доступны объекты охватывающего блока, если только они не экранированы омонимичными (одноименными) локальными переменными. Экранирование означает приоритет локальных объектов над глобальными и является средством разрешения конфликта имен. При использовании блочной структуры управление памятью локальных данных блоков происходит следующим образом: для каждого из блоков на время его выполнения выделяется область памяти для размещения локальных переменных и констант. Глобальные переменные и константы при этом уже размещены в области памяти охватывающего блока. После завершения выполнения блока память, отведенная под локальные переменные и константы, освобождается и может быть повторно использована, что сокращает общую потребность программы в памяти, так что она значительно меньше суммарной длины всех переменных и констант.

3. Цезарь на НАМ

(для входного алфавита - русского, для английского аналогично)

`*A -> D*`

`*B -> E*`

`*C -> F*`

`*D -> G*`

`*E -> H*`

`*F -> I*`

`*G -> J*`

`*H -> K*`

`*I -> L*`

`*J -> M*`

K -> N
L -> O
M -> P
N -> Q
O -> R
P -> S
Q -> T
R -> U
S -> V
T -> W
U -> X
V -> Y
W -> Z
X -> A
Y -> B
Z -> C
-> *
* ->.

Билет 2

1) Предмет информатики (стр 7)

Информатика - наука об ЭВМ и обработке информации. Объединяет несколько более-менее независимых дисциплин. Одной из дисциплин является программирование.

Информатику определяют как науку об автоматической обработке информации при помощи ЭВМ:

Наука об осуществляющей с помощью преимущественно автоматических средств целесообразной обработке информации, рассматриваемой как представление знаний и сообщений в технических, экономических, и социальных областях.

Информатика во многом подобна математике. Подобно ей, она изучает законы, созданные человеком, и поддающиеся доказательству в отличие от естественных законов, знание которых всегда сопровождается некой долей неопределенности. Разница же заключается в предмете и подходе - математика обычно имеет дело с теоремами, бесконечными процессами и статистическими соотношениями, а информатика - с алгоритмами, конечными конструкциями, и динамическими соотношениями.

Как гласит программистский фольклор, математика делает то что можно так, как нужно, а информатика - то что нужно так, как можно.

2) Универсальная МТ (стр.107)

Универсальной машиной Тьюринга называют машину Т, которая может заменить собой любую машину Тп. Получив на вход программу и входные данные, она вычисляет ответ, который вычислила бы по входным данным машина Тп, чья программа была дана на вход.

До начала работы на ее ленту записывается последовательность четверок, представляющая программу машины, работу которой необходимо выполнить, и начальные данные.

Универсальная МТ просматривает программу, записанную на ее ленте, и выполняет команду за командой этой программы, получая на ленте вслед за аргументами результат вычислений. Текст программы (в виде последовательности четверок), помещается на ленте слева от аргументов, и, следовательно, при нормированных вычислениях сохраняется неизменным в процессе выполнения программы.

Доказать универсальность некоторой системы - значит показать, что она может моделировать поведение некоторой системы, универсальность которой доказана, либо целого класса систем, который является универсальным (например всех МТ).

Марвин Минский построил машину с 7 состояниями 4-мя буквами алфавита.

3) Проверка палиндрома числа на Си

Справка: Палиндром — число (например, 404), буквосочетание, слово или текст, одинаково читающееся в обоих направлениях. Иногда палиндромом называют любой симметричный относительно своей середины набор символов.

```
#include <stdio.h>
```

```
#include <limits.h>
```

```

int main()
{
    int n, n1, obr, a, flag;
    printf("Enter N = ");
    while (scanf ("%d", &n) != EOF)
    {
        if (sizeof(n) == 4) printf ("n = %d max = %d\n", n, INT_MAX);
        n = (n > 0) ? n : -n;
        n1 = n;
        flag = 1;
        obr = 0;
        while (n1 > 0)
        {
            a = n1 % 10;
            n1 /= 10;
            if (INT_MAX / 10 >= obr) obr = obr * 10 + a;
            else flag = 0;
        }
        if (flag == 1 && n == obr) printf("%d - полиндром\n", n);
        else printf ("flag = %d %d - не полиндром\n", flag, n);
    }
    return 0;
}

```

Билет 3

1. НАМ (стр.57)

В этой алгоритмической модели преобразуются текстовые сообщения. Элементарными тактами обработки алгоритмов Маркова являются замены подслов исходного сообщения на некоторые другие слова. Нормальные алгоритмы Маркова (НАМ) по существу являются детерминистическими (взаимосвязанными) текстовыми именами, которые для каждого выходного слова однозначно задают вычисления и, тем самым, в случае их завершения, порождают определенный результат. Марковская стратегия применения правил заключается в следующем: 1)Если применено несколько правил, то берется правило, которое встречается в описании алгоритма первым; 2)Если правило применимо в нескольких местах обрабатываемого слова, то выбирается самое левое из этих мест. НАМ представляет собой упорядоченный набор правил-продукций – пар слов, соединенных знаком \rightarrow . Слева от этого знака стоит слово, которое заменяется, а справа от этого знака стоит слово, на которое заменяется. Например строка $az \rightarrow z$ заменяет все слова az на z . Пример работы такой программы: на входе $aaazazaz$, на выходе zzz . НАМ не является нормируемым.

2. Понятие файла (стр. 179 либо 207)

Структура файла является обобщением понятия последовательности. Поэтому файлы следует считать «массивами на диске». Компоненты все одного типа и они доступны только путем последовательного прочтения, движения «назад» нет ввиду инерционности электромеханических устройств. Аналог – магнитная лента. Внешние файлы обычно перечисляются в заголовке программы. Они существуют до начала работы программы и/или сохраняются после окончания ее работы. Внутренние файлы, также как и внешние, описываются в программе как файловые переменные. Их время жизни совпадает с временем работы программы. Текстовые – в них только символы, юзается спец. Знак для конца строки. Нетекстовые файлы не предназначены для ввода-вывода и хранят данные непосредственно во внутримашинном представлении.

3. Целочисленное деление в кардинальной сс на Си. (НА ДТ)

Работаем с текстом (типа вертикальных палочек)

/*Целочисленное деление в кардинальной системе счисления на Си

Ввод : Палочки для двух чисел

Выход: Палочки через пробел*/

#include <stdio.h>

```

#include <stdlib.h>
int main() {
    int i = ' ', flag = 0, a = -1, b = -1, c;
    while((i = getchar())&&(i != EOF)) {
        if (i == (int)'|' && flag == 0) {
            a++;
        }
        if (i == (int)' ' && flag == 0) {
            flag = 1;
        }
        if (i == (int)'|' && flag == 1) {
            b++;
        }
    }
    c = a / b;
    printf("a=%d\nb=%d\nc=%d\n", a, b, c);
    for (int i = -1; i < c; i++) {
        printf("| ");
    }
}
return 0;
}

```

4 билет

1) II теорема Шеннона. Доказательство

Для любой МТ $T=(A,Q,P,q_0)$ можно эффективным образом построить МТ $T''=(A',Q',P',q_0')$, моделирующую машину T и имеющую всего однобуквенный алфавит $A'=\{a_1\}=\{|$.

Доказательство теоремы основывается на следующих рассуждениях. Первоначально последовательность $X=\{x_i\}$ кодируется символами из B так, что достигается максимальная пропускная способность (канал не имеет помех). Затем в последовательность из B длины n вводится r символов по каналу передается новая последовательность из $n+r$ символов. Число возможных последовательностей длины $n+r$ больше числа возможных последовательностей длины n . Множество всех последовательностей длины $n+r$ может быть разбито на $n+r$ подмножеств, каждому из которых сопоставлена одна из последовательностей длины n . При наличии помехи на последовательность из $n+r$ выводит ее из соответствующего подмножества с вероятностью сколь угодно малой.

Теорема позволяет определять на приемной стороне канала, какому подмножеству принадлежит исаженная помехами принятая последовательность $n+r$, и тем самым восстановить исходную последовательность длины n .

Как и в случае Теоремы I, доказательством будет схема построения. Покажем, что можно построить машину **C**, работающую подобно любой заданной машине Тьюринга **A** и использующую только два символа внешний алфавит, например символы 0 и 1.

Пусть машина **A** содержит:

- n внутренних состояний S_j
- m символов внешнего алфавита a_p

Тогда машина **C** будет содержать:

- n внутренних состояний T_j , являющихся аналогами S_j
- не более чем $8nm$ специальных внутренних состояний,
- 2 символа внешнего алфавита: 0 и 1

Общая идея построения

Пусть l - наименьшее целое число, для которого $m \leq 2^l$. Тогда символам машины **A** можно сопоставить двоичные последовательности длины l таким образом, что различным символам будут соответствовать различные же последовательности. При этом пустому символу машины **A** мы ставим в соответствие последовательность из l нулей. Машина **C** будет работать с двоичными последовательностями. Элементарная операция машины **A** будет соответствовать в машине **C** переходу считающей головки на l

- 1 клеток вправо (с сохранением считанной информации во внутреннем состоянии головки), затем обратному переходу на / - 1 клеток влево, записи новых символов по пути и, наконец, движению вправо или влево на / клеток, в соответствии с движением считающей головки машины **A**. В течение этого процесса состояние машины **A**, конечно, сохраняется и в машине **C**. Замена старого состояния новым происходит в конце операции считывания.

2) Тип логический(стр.143)

Boolean. Тип данных, который может принимать всего два значения, 0 или 1, True или False, T или F, ещё неопределённость. Занимает всего 1 бит памяти (но не всегда - в Си размер bool 1 байт). Популярные операции: конъюнкция, дизъюнкция, тождественность и отрицание. Операции логического типа обладают следующими свойствами:

1. Если хотя бы один операнд имеет значение NULL, то и результат имеет значение NULL

2. X&X=X, XVX=X при X=И, Л

3. X = И, Y=любое значение кроме NULL, то, XXY=И

4. X = Л, Y=любое значение кроме NULL, то, X&Y=Л

5. Дизъюнкция и конъюнкция коммутативны.

6. XV-X=И, кроме X= NULL

7. X&-X=Л, кроме X= NULL

8. $\neg(\neg X)=X$

Можно нарисовать таблицы истинности для 4-х популярных операций, а также диаграммы тьюринга

NULL - неопределённость

3) Сложение в троичной системе счисления на ДТ

5 билет

1)Знаки и символы (стр.18)

Письменные языковые сообщения представляют из себя последовательности знаков. Мы будем различать атомарные и составные знаки. Атомарным знаком или буквой называется «неделимый» символ 1-ого уровня. Пример: буквы (А, Б, В....). Они складываются в знаки второго уровня (слова). Знаки второго уровня в знаки третьего (предложения) и т.д. (абзацы и пр.). Все знаки большего 1 уровня называются составными знаками. Множество слов также можно рассматривать как набор знаков. Для записи сообщения, которое содержит знаки k-го уровня необходимо иметь k-1 различных типов знаков разделителей. Есть специальный знак, чтобы разделять одни составные знаки от других. Для слов это пробел, для предложений – точка. С каждым знаком связывается его смысл или семантика. Знак вместе с сопоставленной ему семантикой называется символом. (А – буква, пи – символ).

2) Критика моделей вычислений Тьюринга(стр. 113)

Недостатки МТ довольно существенны.

1. Описание программы, выполняющая алгоритм сложения двух дробей занял целую страницу, поэтому для более сложных математических операций понадобится большая трудоемкость программиста, даже при составленном алгоритме.

2. Необходимость многочисленных копирований. При описании нашего алгоритма около половины всех действий были именно действиями копирования. Из-за этого поиск нужных слов среди всего этого громоздкого кода становится довольно затруднительным.

3. Необходимость при составлении программы выписывать ситуации на ленте, иначе можно легко запутаться. МТ удобно для построения красивой теории алгоритмов, но не для практических задач.

Написать, что Фон Нейман создал машину, усовершенствовав МТ.

3)Вывести первые 500 простых чисел на Си

```
#include <stdio.h>
int main(){
    int i=2, k, m, n=500;
    while (n>0){
        m=i;
        k=0;
        while (m>1){
```

```

    if (i%m==0) k++;
    if (k>1) break;
    m--;
}
if (k==1) {
    printf ("%d ",i);
    n--;
}

```

Билет 6

1. Информация и сообщение (стр.16)

Информация и сообщение – основные (неопределяемые) понятия информатики. Информация передается посредством сообщения и наоборот, сообщение – то, что несет информацию. Информация может существовать только в форме некоторого сообщения. Соответствие между информацией и несущим ее сообщением не является взаимно однозначным: 1) Одна и та же информация может передаваться с помощью различных сообщений; 2) одно и то же сообщение может передавать различную информацию

2 . Обобщенная инструкция композиции(стр.131)

Общепринятый и широко используемый способ организации вычислений громоздких выражений – *функциональная композиция*, состоящая в том, что значения одной или нескольких функций используются в качестве аргументов для других функций.

Поясним это на примере.

Пусть требуется вычислить корни квадратного уравнения $ax^2 + bx + c = 0$. Формулы для вычисления корней квадратного уравнения $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ удобно для вычислений представить в виде $x_{1,2} = \frac{-b \pm d}{e}$, где $d = \sqrt{b^2 - 4ac}$ и $e = 2a$. (Будем пока считать, что проблемы с вычислением квадратного корня из дискриминанта не существует.) Таким образом, вместо исходного соотношения для вычисления корней квадратного уравнения используется другое соотношение, аргументами которого являются значения двух выделенных выражений.

BEGIN

Инструкции описания объектов программы: коэффициентов уравнения a , b , и c , корней уравнения x_1 и x_2 , дискриминанта d и переменной e для хранения значения удвоенного первого коэффициента

Инструкции ввода начальных значений коэффициентов уравнения: a , b и c

```
e := 2.0 * a;
d := SQRT(b * b - 4.0 * a * c);
x1 := (-b + d) / e;
x2 := (-b - d) / e;
```

Инструкции вывода: a , b , c , x_1 , x_2

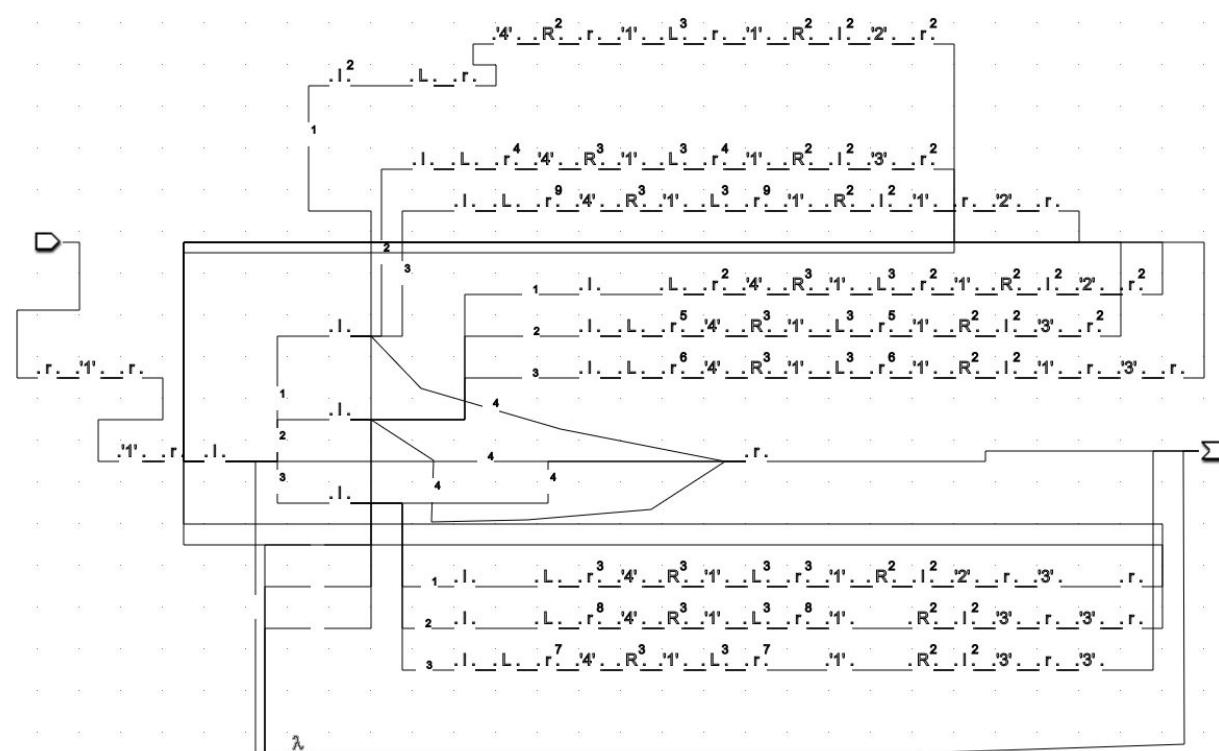
END

В языке Паскаль последовательная композиция инструкций скрепляется т. н. операторными скобками **begin** и **end** в Паскале и символами { и } в Си. Полученная таким образом составная инструкция может быть использована всюду, где дозволяется простая инструкция.

В настоящее время, когда параллельное выполнение, многопроцессорность и многозадачность реальны и доступны, нельзя не упомянуть и о параллельной композиции. Композируемые элементы в этом случае выполняются параллельно и независимо, что должно отражаться в форме записи: вместо точки с запятой для разделения параллельно выполняемых ветвей могут использоваться знаки || или □.

3. Транспонирование матрицы 3*3 на ДТ

Один ("1") в кавычках - метасимвол А, 4 = * как то так. За 100% работу не ручаюсь



7 билет

1. Эквивалентность диаграмм и программ(стр.54)

Каждой ДМТ, задающей программу Р, можно эффективным образом сопоставить МТ $T=(A,Q,P,q_0)$, образованную строками команд МТ так, чтобы МТ, определяющая эту диаграмму, смоделировала бы машину Т. Т.е. нам нужно указать способ эффективного построения программы по ДМТ программы Р, а потом убедиться, что для исходной ДМТ и МТ, выполняются все пять условий моделирования МТ. Построение: Каждому элементу диаграммы сопоставим строку команды А ,Q, P, qо ,при этом нам нужно будет поставить вместо последней точки ДМТ заключающую строку программы МТ. Док-во, что оно эквивалентно. Все 5 условий моделирования должны быть выполнены. Условие 1 выполняется потому, что алфавиты МТ и ДМТ совпадают. Условие 2 выполняется потому, что точкам ДМТ соответствуют состояния МТ. Условие 3-5 выполняются потому, что МТ определяет ту же последовательность элементарных действий, что и программа Р.

2. Структура программ для машины фон Неймана(стр.123)

Программа и данные помещаются в одном и том же устройстве памяти, но на регистрах процессора слова с командами программы и данными существенно различаются. Таблица имен, программа и данные могут быть записаны в виде, непосредственно переносимом в память машины. Процесс составления таблицы имен и размещения данных в памяти машины может быть автоматизирован. Каждый объект, используемый для получения значений других объектов при выполнении программы обработки данных должен иметь определенное значение. Начальные значения таких объектов могут быть заданы либо при размещении в их памяти машины (константы), либо путем выполнения инструкции ввода данных. А теперь к сути. Программа для машины фон Неймана представляет собой текст, обязательно включающий в себя инструкции описания объектов (данных, которые обрабатывает программа) инструкции ввода данных, инструкции обработки данных и инструкции вывода значений объектов-результатов. Начало => инструкции описания объектов программы => инструкции ввода => инструкции обработки данных => инструкции вывода => КОНЕЦ

3. Реверс 111 на Си (Реверс байтов целого числа) by sakost

```
#include <stdio.h> //standart input-output library including
#include <stdlib.h> //standart library including

//gets byte and reverse it
__int8 byte_reverse(__int8 intitial_byte)
{
    __int8 reversed_byte = 0;           //intitilization of reversed_byte
    __int32 cnt = sizeof(__int8) * 8; //intitilization of counter, which contains
quanity of bites in byte
    while (cnt)                      //cycle for all bites in byte
    {
        reversed_byte <<= 1;          //shift to begin of notation
        reversed_byte |= (intitial_byte & 1); //write to notation
        intitial_byte >>= 1;          //shift to end of notation
        cnt--;                      //couter decrement (postdecrement)
    }
    return reversed_byte; //returns reversed byte
}

main (void){
__int32 proc_int, res_int = 0; //declaration of processed integer and initialization of
copy of resulting integer
    scanf("%d", &proc_int);           //read proc_int from stdin

    for (__int32 i = 0; i < 4; ++i) //cycle for all bytes
    {
        res_int >>= 8;                //shift to
end of notation
        res_int |= ((__int32)(byte_reverse((__int8)(proc_int & 0xFF))) << 24); //write to
notation
    }
}
```

```

    proc_int >>= 8;                                //shift to
end of notation
}

printf("%d", res_int); //write result to stdout

return 0; //return 0 to caller
} function which gets integer number from keyboard, and revers it's bytes in 4-bytes
notation
int main()
{
    __i
}

```

/*-/ 8 билет

1) Системы счисления(стр.24)

Система счисления – это способ числовой интерпретации цифровых сообщений. Непозиционные: а) Натуральная система счисления. Число принадлежит целым от единицы до бесконечности. Пример – пересчет предметов. II яблока. б) Кардинальная система счисления. Для нуля отдельная палочка. Недостатком этих систем счисления являются - громоздкость! Или, например, усложняется операция сложения: надо убрать одну лишнюю палочку для правильного представления суммы.

Проблема: для числа 1000 потребуется половина экрана терминала.

Позиционные: $A_n = a_k * n^k - 1 + \dots + a_2 * n^1 + a_1 * n^0$. Для отрицательных чисел вся эта сумма домножается на -1. Пример (десятичная) $349_{10} = 3 * 10^2 + 4 * 10^1 + 9 * 10^0$. Перевод из двоичной в десятичную. $101_2 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0$. Как мы видим, позиция влияет на значение числа. Поэтому эта система счисления называется позиционной

2) Построение процессора фон Неймана(стр.118)

Таким образом, предварительно разработав и записав на D -ленте набор внешних МТ, реализующих действия, используемые при составлении некоторого класса тьюринговых программ, мы получаем *процессор* для этого класса программ, позволяющий существенно упростить описания программ рассматриваемого класса и сократить процесс их разработки. Действия, реализуемые МТ, записанными на D -ленту, будем называть *элементарными действиями процессора или операциями*. Все внешние МТ, реализующие операции, имеют одинаковый рабочий алфавит, называемый *рабочим алфавитом процессора*. Этот алфавит является рабочим алфавитом всех программ, выполняемых на процессоре.

Для построения процессора необходимо:

1. Выбрать и зафиксировать рабочий алфавит **A** процессора. Обычно это первые p неотрицательных чисел, включая ноль. Т.е. от 0 до $p-1$.
2. Зафиксировать конечное или бесконечное множество допустимых слов над алфавитом. Если множество допустимых слов конечно и включает лишь слова фиксированной длины **k** (либо слова, имеющие длину не больше, чем **k**), то удобно добавить к множеству допустимых слов еще одно слово, называемое «переполнением».
3. Составить и записать на ленту *программы* МТ, выполняющие операции процессора, включая операции РИМ (значение переменной) и АДР (адрес переменной)
4. Сообщить обозначения операций универсальной МТ, точнее, управляющей программе универсальной МТ, на основе которой строится процессор.

3) Поразрядная конъюнкция двоичных чисел на ДТ

1 вариант(ВНИМАНИЕ! Данные алгоритм придуман 09.03.19 и, на данный момент, не проверен. Доверяйте источнику на свой страх и риск. Удачи на сдаче!):

На вход подаются два двоичных слова одинаковой длины, написанных через *лямбда*(пробел), &(конъюнкция), *лямбда*(пробел). Пример 10001_&_11101(Вместо знака "_" - пробел)

В конце выполнения программы на ленте должны остаться исходные данные и третье двоичное число, которое является результатом поразрядной конъюнкции.

Алгоритм:

1)С помощью набора команд L,I,I,L(или L,L,L) мы оказываемся слева от первой цифры первого числа(Довольно простой алгоритм для начала).

2)Далее с помощью г переходим к первой цифре(Далее начинается разветвление).

2.1.0)Если в ячейке находится 1, то мы заменяем её на A, и, пропуская все оставшееся слово, переходим ко второму числу. Тут пропускаем все возможные A, B, мы натыкаемся на цифру(Снова разветвление(не обижайтесь)).

2.1.1) Если в ячейке 1, то она заменяется на A, и(снова), пропуская все оставшееся слово, доходим до первого после слова "пробела", делаем еще один шаг вправо. Если есть число/буква(хотя буква не будет в третьем слове), мы пропускаем их всех и в пустую ячейку после части слова вписываем 1-цу, делаем команду **Back**.

//**Справка:** Back - это "ветка внутри дерева"(Не ясно, да?:)), которая отвечает за наше возвращение в начало первого слова(Выглядит примерно так: НАЧАЛО->L->L->I->I->L->КОНЕЦ)

2.1.2) Если в ячейке 0, то она заменяется на B(Далее по аналогии с 2.1.1, кроме того, что в третье слово мы вписываем 0, а не 1).

2.2.0)Делаем по аналогии с 2.1.0(Замена на B)

2.2.1)Если в ячейке 1, то она заменяется на A, и, пропуская все оставшееся слово, доходим до первого после слова "пробела", делаем еще один шаг вправо. Если есть число/буква, мы пропускаем их всех и в пустую ячейку после части слова вписываем 0, делаем команду **Back**.2.2.2)Если в ячейке 0, то она заменяется на B(Далее по аналогии с 2.2.1)

2.3)(Самое вкусное, END GAME) Если мы увидим пробел, то(неожиданно) мы осознаём, что слово закончилось, и оно полностью состоит из A/B. Теперь наша задача, заменить все буквы A и B на 1 и 0 соответственно. Для этого мы, с помощью L, переходим в начало первого слова, и начинаем цикл, который мы отметим как **Reverse**

9 билет

1) обработка сообщений(стр.29)

Примеры: кодирование, перевод с одного языка на другой, редактирование текста, решение системы уравнений чтение вслух и пр. Т.е. обработка сообщений состоит в выделении в исходном сообщении знаков некоторого уровня, составляющих это сообщение, и замене каждого выделенного знака другим знаком. При чтении вслух каждый слог заменяется фонемой; последовательность фонем составляет результирующее устное сообщение. Следовательно любая обработка сообщений может рассматриваться как кодирование в широком смысле.

2) обобщенная инструкция присваивания(стр.126)

Согласно одному из принципов фон Неймана вычислительная машина должна осуществлять покомандную обработку данных. Операнды каждой команды вызываются на регистры из памяти, после чего эта команда выполняет необходимое действие над ними, оставляя результат также на регистрах. Для сохранения результата он должен быть скопирован в память, поскольку регистры требуются для следующей команды. Поэтому программная версия машины фон Неймана должна содержать аналог цепочки «загрузка операндов — выполнение команды — сохранение результата». В любом языке программирования фон Неймановского типа основной элементарной инструкцией обработки данных является **инструкция присваивания**, которая и является таким аналогом. Инструкция присваивания имеет вид

X := A;

где символ := является знаком присваивания; X обозначает имя объекта (простой, индексированной или квалифицированной переменной, либо, иногда, массива), A — выражение, задающее последовательность действий над операндами. В языке Си все, что может стоять в левой части присваивания, называется *lvalue*. В простейшем случае, когда X и A — простые переменные или константы, действие инструкции присваивания сводится к перезагрузке значения переменной A в переменную X через регистр. В явном виде это выражается так

X := АДР(РИМ(A))

что имеет следующий императивный смысл: сделать адрес объекта с именем X адресом значения объекта A.

3) НАМ проверка лексикографического упорядочивания двоичных слов

на вход поступают двоичные числа равной длины через символ _

(При проверке на эмуляторе НАМ, прога зависает в самом начале, т.к. у проги нет строки, которая могла бы начать процесс(мб не хватает какого-нибудь символа в исходных данных))

```
#_->#
#&->#*
*0&->0*
*1&->1*
*0_->0_
*1_->1_
*1->/
```

```
*0->/  
/1->/  
/0->/  
/_->/  
_->/  
1/->/  
0/->/  
#/->/  
/->.YES  
%1->%  
%0->%  
%_>%  
_%->%  
1%->%  
0%->%  
%&->%  
&%->%  
%->.NO  
#0->  
#1->@  
@0->0@  
@1->1@  
@_>_!  
>0->0  
>1->1  
>_>_  
^&0->&0^  
^&1->&1^  
^0-><&0  
^1-><&1  
!&0->&0!  
!&1->&1!  
!0->%  
!1->?&1  
&0<-><&0  
&1<-><&1  
&0?->?&0  
&1?->?&1  
_?->?  
_-<->_  
0<-><0  
1<-><1  
<->#  
0?->?0  
1?->?1  
?->#  
&0->0  
&1->1
```

10 билет:

1) Свойства алгоритмов(стр.34)

1. Массовость, т.е. потенциальная бесконечность исходных сообщений для обработки
2. Детерминированность (определенность) – заключается в последовательном применении шагов, однозначно определяемых результатами каждого предыдущего шага
3. Элементарность каждого шага, каждый шаг алгоритма должен выполнять только одно определенное действие
4. Результативность алгоритма - точное описание того, что считается результатом применения алгоритма после выполнения всех требуемых шагов.
5. Сложность алгоритма определяется кол-вом простых операций, которые нужно совершить для обработки сообщения. От него зависит время выполнения алгоритма.

2) Построение машины фон Неймана (самовоспроизводящиеся машины)(стр.121)

Машиной фон Неймана называется аппаратная реализация процессора фон Неймана. Машина состоит из управляющего устройства, устройства памяти и одного или нескольких регистров. Память машины фон Неймана содержит конечное число ячеек ограниченного размера. В состав машины должны входить устройства записи сообщений в память и вывода данных из памяти на носитель, доступный для восприятия органами чувств человека. В результате аппаратной реализации процессора фон Неймана мы перешли от сообщений к данным, потеряв абсолютную вычислимость и лишившись возможности непосредственно созерцать процесс обработки данных в ЭВМ и тем более участвовать в нём.

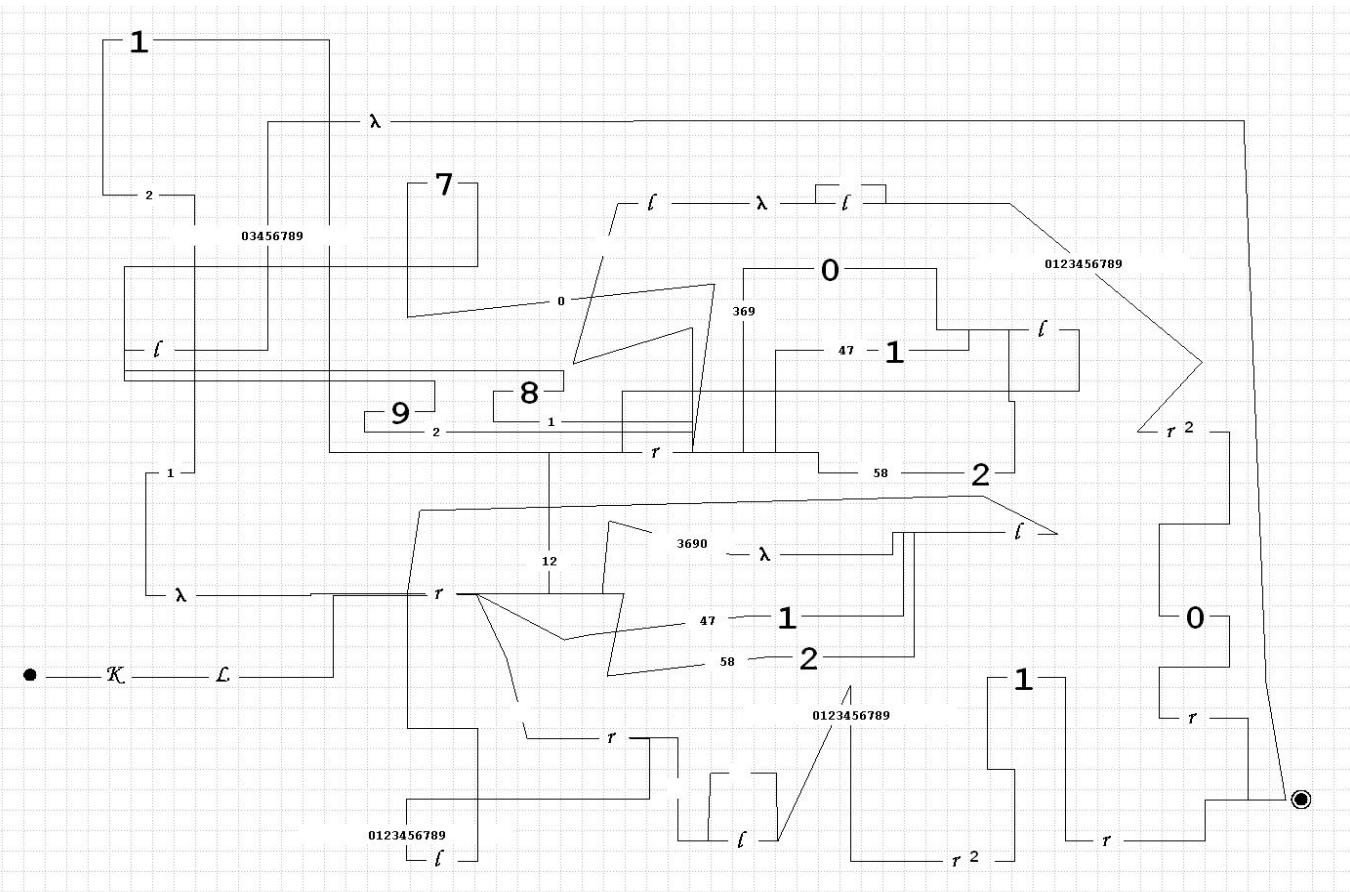
Но приобретения более значительны: мы получили возможность быстрой и безошибочной автоматической обработки данных без участия человека. Абсолютная вычислимость в машине фон Неймана заменена частичной. Вычислимым по фон Нейману считается все то, что может быть получено за приемлемое время на доступных ресурсах памяти и, весьма часто, в пределах ограниченного бюджета имеющихся средств.

3) ДТ проверки делимости на 3

(100% рабочая при нормальных исходных данных)

краткое устное описание:

копируем число и дальше начинаем работать с копией--> начинаем обрабатывать число со старших разрядов и постепенно двигаемся к младшим, в зависимости от цифры на которой сейчас находится каретка, программа или заменяет цифру на новую или переходит на более младший разряд. обрабатывает этот разряд и обнуляет предыдущий (если это требуется).--> программа продолжает работу до тех пор пока не дойдет до последнего разряда. обнуляет его и возвращает или 1 (истина) или 0 (ложь)



11 билет:

1) обработка информации(стр.30)

Во всех случаях, когда правило обработки σ является отображением, правило обработки сообщений v называется сохраняющим информацию. Если v сохраняет информацию, то диаграмма коммутативна: $v \circ \varphi = \varphi \circ \sigma$. Отображение σ называется в таком случае правилом обработки информации.

Если σ обратимое отображение, т.е. информация при обработке не теряется, то соответствующая обработка v называется першифровкой.

Пусть ν – обратимая пересифровка. Тогда обработка обратимой пересифровкой называется перекодировкой. Иначе она называются сжимающей. Т.е. в результате необратимой пересифровки сообщений их кол-во уменьшается, а информация может либо сохраняться, либо теряться.

Во всех случаях, когда соответствие σ является отображением, правило обработки сообщений ν называется **сохраняющим информацию**. Если правило обработки сообщений ν сохраняет информацию, то диаграмма

$$\begin{array}{ccc} N & \xrightarrow{\varphi} & I \\ \nu \downarrow & & \downarrow \sigma \\ N' & \xrightarrow{\varphi'} & I' \end{array} \quad (**)$$

коммутативна: $\nu \circ \varphi' = \varphi \circ \sigma$. Отображение σ называется в этом случае **правилом обработки информации**.

Обычно обработку информации сводят к обработке сообщений, т. е., исходя из требуемого правила обработки информации σ , пытаются определить отображения ν , φ и φ' таким образом, чтобы диаграмма $(**)$ была коммутативной.

Если σ – обратимое (взаимно однозначное) отображение, т. е. если информация при обработке по правилу σ не теряется, то соответствующую обработку сообщений ν называют **пересифровкой**.

Пусть ν – обратимая пересифровка. Тогда по сообщению $n' = \nu(n)$ можно восстановить не только исходную информацию, но и само исходное сообщение n . Иными словами, в этом случае n' **кодирует** n (см. п. 1.4). Обратимая пересифровка ν называется **перекодировкой**.

Пусть пересифровка ν не является обратимой, т. е. пусть несколько сообщений из N кодируются одним и тем же сообщением из N' . Но так как при пересифровке информация не теряется, это означает, что исходное множество сообщений N является избыточным: некоторые сообщения из N содержат одну и ту же информацию (дублируют друг друга). В N' таких дублирующих сообщений меньше, чем в N , так как при обработке по правилу ν некоторые из дублирующих друг друга сообщений «сливаются» в одно сообщение. Пересифровка ν , которая не является обратимой, называется **сжимающей**. Сжатию подвергается множество сообщений. То есть в результате необратимой пересифровки сообщений их количество уменьшается, а информация может либо сохраняться, либо теряться.

2) Специализированные процессоры для обработки сообщений (стр.119)

Универсальная МТ, D -лента которой содержит программы, реализующие арифметические действия, имеет **качественно новый уровень** по сравнению с универсальной МТ с пустой D -лентой: алгоритмы, связанные с выполнением арифметических действий над целыми числами (например, алгоритм Евклида), описываются на ней гораздо более простыми программами. Чтобы подчеркнуть это обстоятельство, назовем такую универсальную МТ **арифметическим процессором**.

3) НАМ: умножение двух чисел в натуральной системе счисления

1) 1111#11111 через решетку два т числа

```
1a->ab1
ab->b
1b->b1
#1->a#
1#->#
#->c
bc->c1
c->
```

2) вариант из методички

```
|>*<->>*<d
d| -> |md
dm -> md
d>->
<>*<-><e
e| -> e
```

em -> |e
e> -> >

12 билет

1) Сложность алгоритмов(стр.36)

Сложность алгоритма определяет, как долго придется ждать, пока будет получен результат его работы. Всего выделяют 7 классов сложности:

- 1) O (1) – алгоритмы с постоянным временем выполнения, например доступ к элементу массива. При увеличении размера задачи вдвое время выполнения не меняется
- 2) O ($\log n$) – Алгоритмы с логарифмическим временем выполнения, например, бинарный поиск. Время выполнения удваивается при увеличении размера задачи в n раз.
- 3) O (n) - Алгоритмы с линейным временем выполнения, например, последовательный поиск или схема Горнера. При увеличении размера задачи вдвое, время выполнения также удваивается.
- 4) O ($n^{\log(n)}$) - Алгоритмы с линеарифмическим временем выполнения, например, быстрая сортировка. При увеличении размера вдвое, время выполнения увеличивается немногим больше, чем вдвое.
- 5) O(n^2) - Алгоритмы с квадратичным временем выполнения, например, простые алгоритмы сортировки. При увеличении размера задачи вдвое, время выполнения увеличивается в 4 раза.
- 6) O (n^3) – Алгоритмы с кубическим временем выполнения, например, умножение матриц. Время выполнения увеличивается в 8 раз при увеличении размера задачи в 2 раза.
- 7) O (2^n) - Алгоритмы с экспоненциальным временем выполнения, например, задача о составлении расписания. Время выполнения увеличивается в 2^n при увеличении размера задачи в 2 раза

2)Нормированные функции, теорема(стр.81)

Определение 2.5.2. Функция $f : (A_s^*)^n \rightarrow A_t^*$ называется **нормированно вычислимой по Тьюрингу** (НВТ-функцией), если существует МТ T_f , которая вычисляет f и при этом удовлетворяет следующим требованиям:

1. если $X = (u_1, u_2, \dots, u_n) \notin Def(f)$, то машина T_f после применения к X никогда не останавливается;
2. если $X = (u_1, u_2, \dots, u_n) \in Def(f)$, то T_f вычисляет значение функции

$$[\lambda z \lambda u_1 \lambda u_2 \dots \lambda u_n (\lambda) \lambda] \xrightarrow{T_f}^* [\lambda z \lambda u_1 \lambda u_2 \dots \lambda u_n \lambda f(u_1, u_2, \dots, u_n)(\lambda) \lambda], z \in \bar{A}_p^* \text{ и останавливается.}$$

Смысл нормированного вычисления состоит в том, что при таком вычислении часть ленты, расположенная левее символа λ , непосредственно предшествующего u_1 , не меняется и не влияет на работу машины T_f . При нормированном вычислении головка не должна заходить левее указанного символа λ .

3) написать Дифференциал многочлена на си

//Дифференциал многочлена (на вход поступают
//коэффициенты многочлена, на выход - коэффициенты
//производной от многочлена)
#include <stdio.h>

```
int main(){  
    int a[100];  
    int power = -1;  
  
    while(power > 100 || power < 0){  
        printf("Введите степень многочлена:\n");
```

```

    scanf("%d",&power);
}

printf("Введите коэффициенты: \n");
for(int i = 0; i <= power; i++)
    scanf("%d",&a[i]);

printf("Результат:\n");
for(int i = 0; i < power; i++)
    printf("%d ", a[i]*(power-i));
printf("0\n");
}

```

Билет 13

1) Конструктивное описание процесса обработки дискретных сообщений(стр.32)

из сообщения $d \in D$. Если D — конечное множество, то отображение P может быть задано таблицей, в которой перечисляются все сообщения $d \in D$ и соответствующие им сообщения $P(d)$. Примером такой таблицы может служить таблица умножения (или таблица сложения) однозначных чисел в позиционной системе счисления. Если же D бесконечно или по крайней мере так велико, что задание P с помощью таблицы оказывается непрактичным, то нужно представить P в виде последовательности элементарных **шагов обработки** (или **тактов**), каждый из которых состоит в выполнении одного или нескольких достаточно простых отображений, называемых **операциями**: $P = P_1 \circ P_2 \circ \dots \circ P_k$. Примеры операций: переписывание (копирование) буквы или слова, приписывание буквы к слову, приписывание слова к другому слову. Отметим, что любое отображение, допускающее описание с помощью достаточно простой таблицы, может быть объявлено операцией. Конечно же, оператор языка программирования высокого уровня, машинная команда, директива или системный вызов операционной системы также удовлетворяют критерию элементарности шагов обработки.

Обработка (дискретных) сообщений состоит в применении отображения (правила обработки) ν (см. п. 1.6), или в последовательном применении отображений C , P и Q (см. п. 1.8).

Чтобы отображение P могло служить основой для автоматизации обработки сообщений, оно должно задавать некоторый **способ построения** сообщения $d' = P(d)$, исходя

Пример 1.9.1. Пусть исходное сообщение — это пара (u_1, u_2) слов над некоторым (например, латинским) алфавитом, к которому перед буквой a добавлен пробел, и пусть требуется, чтобы результирующее сообщение (w_1, w_2) состояло из тех же слов, но расположенных в лексикографическом порядке. Правило обработки может быть задано следующей последовательностью элементарных шагов:

- (1) завести пустые слова w_1 и w_2 (это можно сделать, например, написав $w_1 = ''$ и $w_2 = ''$), уравнять количество букв в словах u_1 и u_2 , добавив в конец более короткого слова пробелы и перейти к шагу (2);
- (2) сравнить (пользуясь списком букв в алфавитном порядке, или таблицей для отношения алфавитного предшествования \prec) $\ell(u_1)$ и $\ell(u_2)$ ($\ell(u)$ — обозначение для крайней левой буквы слова u , отличной от пробела); если $\ell(u_1) \prec \ell(u_2)$, перейти к шагу (3); если $\ell(u_2) \prec \ell(u_1)$, перейти к шагу (4). Если буквы $\ell(u_1)$ и $\ell(u_2)$ совпадают и не являются пробелами, то присвоить $\ell(u_1)$ (или $\ell(u_2)$) к словам w_1 и w_2 . Заменить $\ell(u_1)$ и $\ell(u_2)$ пробелами и снова выполнить шаг (2). Наконец, если $\ell(u_1)$ и $\ell(u_2)$ — пробелы, закончить обработку сообщения;
- (3) присвоить слово u_1 к слову w_1 , слово u_2 к слову w_2 и закончить обработку сообщения;
- (4) присвоить слово u_1 к слову w_2 , слово u_2 к слову w_1 и закончить обработку сообщения.

2)Обобщенная инструкция ветвления(стр.134)

3.1.5 Обобщенная инструкция ветвления

Инструкцию ветвления (**IF–FI**) определим следующим образом:

```
IF <предохранитель1> <охраняемая инструкция1>
  ┌ <предохранитель2> <охраняемая инструкция2>
  ...
  ┌ <предохранительm> <охраняемая инструкцияm>
FI
```

Символы **IF** и **FI** играют роль открывающей и закрывающей скобок инструкции ветвления, символ **└** играет роль разделителя охраняемых инструкций, входящих в состав ветвления.

По инструкции ветвления из списка охраняемых инструкций, заключенных в скобки **IF–FI**, должна быть выбрана одна и только одна инструкция (для выполнения), и причем та, предохранитель которой принимает значение **И** (истина).

С использованием инструкции ветвления программа для вычисления значения функции (см. п. 3.3.2) может быть записана следующим образом:

```
BEGIN
  ...
  IF X ≥ 0 ? f := X;
  ┌ X < 0 ? f := X + X * X;
  FI;
  ...
END
```

Правила выполнения инструкции ветвления следующие:

- Одновременно и независимо вычисляются все предохранители.
- Среди вычисленных предохранителей инструкции ветвления должен быть хотя бы один, принимающий значение **И**. Если среди предохранителей инструкции ветвления нет ни одного, принимающего значение **И**, то происходит ОТКАЗ – выполнение программы прекращается (аварийно!). Таким образом, отсутствие среди предохранителей инструкции **IF–FI** предохранителя, принимающего в момент выполнения инструкции значение **И**, воспринимается как грубая ошибка периода выполнения, препятствующая дальнейшему хоть сколько-нибудь осмысленному выполнению программы. Дело в том, что при планировании ветвления программист должен позаботиться о «полноте» системы предикатов в охраняемых инструкциях, и предусмотреть дополнительную ветвь, отрицающую все предикаты.
- Допускается, чтобы среди предохранителей инструкции ветвления было более одного предохранителя, принимающего значение **И** при выполнении инструкции **IF–FI**.

При этом не предполагается, что охраняемые инструкции каким-либо образом упорядочены. Следовательно, если среди предохранителей инструкции **IF–FI** окажется, например, два предохранителя, принимающих значение **И** (истина), то выбор инструкции для выполнения осуществляется недетерминированным образом и не определяется порядком записи охраняемых инструкций в инструкции **IF–FI**. Недетерминированное ветвление,

вообще говоря, неалгоритмично и отражает тот факт, что программисту все равно, по какой из открывшихся ветвей действительно произошло ветвление. Недетерминированность не следует считать случайностью, т. к. случайные величины подчиняются вероятностным законам, что противоречит духу недетерминированности. Недетерминированное ветвление отражает стремление к повышению семантического уровня языка программирования до пропозициональной семантики.

Все известные конструкции ветвления в языках программирования являются частными случаями обобщенного ветвления Э. Дейкстры.

Например, двузвенное ветвление

IF p ? s1 not p ? s2 FI

в Паскале и Си реализуются практически одинаково:

if p then s1 else s2;

if(p) s1(); else s2();

а многозвездный переключатель (избыточный по теореме Бойма-Джакопини-Миллса):

IF e = e1 ? s1
 | e = e2 ? s2
 | . . .
 | e = en ? sn
FI

записывается на этих языках несколько по-разному:

case e of
 e1 : s1;
 e2 : s2;
 { ... }
 en : sn;
end;

switch(e)
{
 e1: s1(); break;
 e2: s2(); break;
 // ...
 en: sn();
}

Естественно, здесь все детерминировано. Пользуясь конкретными вариантами инструкции ветвления, необходимо внимательно изучать их особенности и не только по описанию стандарта языка. Например, в Паскале в случае несовпадения текущего варианта ни с одной из предусмотренных меток вместо благородного и шумного отказа от неудачного ветвления втихую происходит продолжение работы программы. Исправление такой ситуации веткой, выполняемой по умолчанию (otherwise или else), увы, не стандартизовано.

В С/C++ есть стандартная опциональная метка **default**, которая получает управление в случае, когда не сработал ни один из **case**'ов.

3)Проверка палиндрома с помощью НАМ

либо двоичное число

двоичное число

I0->0I

I1->1I

I->m

ygm->YES

yg0m->.YES
yg1m->.YES
yg11->y1b
yg10->y0c
yg01->y1b
yg00->y0c
0g->g0
1g->g1
1@->@
0@->@
sy@->NO
sz@->NO
y@m->.NO
b1->1b
b0->0b
c1->1c
c0->0c
0c->g
1c->@
0b->@
1b->g
a1->y1b
a0->y0c
->al

решение: всего три буквы, а то алфавит очень большой, много кода. Тесты: abba ababa abbc bacb bbb cab.

##a->a##
##b->b##
##c->c##
a(a)->(a)a
b(a)->(a)b
c(a)->(a)c
a(b)->(b)a
b(b)->(b)b
c(b)->(b)c
a(c)->(c)a
b(c)->(c)b
c(c)->(c)c
#(a)a->#
#(b)b->#
#(b)#->##
#(c)#->##
a#->(a)#
b#->(b)#
c#->(c)#
#0->-0
)0->-0
(-0->-0
a-0->-0
b-0->-0
c-0->-0
1###->.1
##->-0
1-0->.0
->1###

14 билет

1. Линейная запись схем МТ(стр.109)

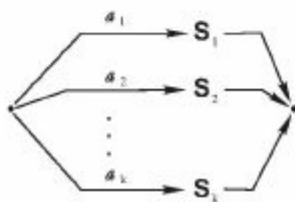
2.8.2 Линейная запись схем машин Тьюринга

Трудность записи схем МТ на ленту состоит в том, что схемы двумерны, в то время как на ленту можно поместить лишь линейную запись. Таким образом, возникла задача линеаризации записи схем. Главным в этой задаче является не запись на ленту «цифровой фотографии» двумерной схемы МТ, а получение структурно-топологического, «векторного» представления. Элементы такого представления мы уже получали при доказательстве эквивалентности диаграмм и программ.

В п. 2.7.2 было отмечено, что схемы допускают всего три вида сочетаний составляющих их МТ: композицию, ветвление и цикл. Рассмотрим, как можно линеаризовать каждое из этих сочетаний.

Композиция состоит в последовательном выполнении нескольких действий, представленных в схеме символами соответствующих МТ. Эта часть записи схемы и так линейна, так как в случае композиции символы указанных МТ просто выписываются один за другим.

Ветвление изображается фрагментом схемы, который имеет вид



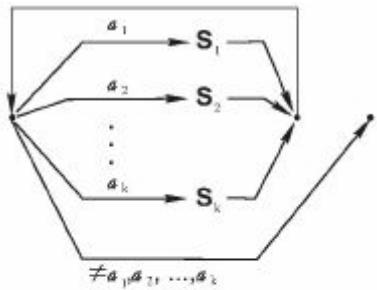
Здесь левая точка описывает состояние q , в котором происходит ветвление. Буквы a_1, a_2, \dots, a_k вместе с состоянием q образуют определяющие пары, указывающие, какая из ветвей должна быть выбрана. Символы S_1, S_2, \dots, S_k представляют собой фрагменты схемы, описывающие, какие действия должны быть выполнены в каждой из ветвей. Правая точка определяет состояние q' после ветвления.

В линейной записи рассматриваемый фрагмент схемы будем представлять в виде

IF $a_1?S_1 \quad \square \quad a_2?S_2 \quad \square \quad \dots \quad \square \quad a_k?S_k \quad \mathbf{FI}$

IF и **FI** соответственно обозначают начало и конец ветвления (состояния q и q' соответственно). Знак «?» отделяет букву, надписанную над стрелкой, от фрагмента схемы, который описывает действия в соответствующей ветви. Символ \square отделяет одну ветвь от другой.

Цикл изображается следующим фрагментом схемы:



Первая и последняя точки в этом фрагменте соответствует состояниям q (начало цикла) и q' (конец цикла). Буквы a_1, a_2, \dots, a_k вместе с состоянием q составляют определяющие пары, по которым цикл должен быть продолжен (при этом выполняется одно из действий, описываемых фрагментами схемы S_1, S_2, \dots, S_k). Буквы, отличные от a_1, a_2, \dots, a_k , вместе с состоянием q формируют определяющую пару, задающую выход

из цикла (переход в состояние q'). Средняя точка тоже соответствует состоянию q (она введена в схему лишь для удобства).

В линейной записи описание цикла имеет вид

DO $a_1?S_1 \square a_2?S_2 \square \dots \square a_k?S_k$ **OD**

DO и **OD** обозначают соответственно начало и конец цикла, а знаки **?** и **□** имеют тот же смысл, что и в случае ветвления.

2. Согласование типов(стр.174)

Подобно многотиповой машине фон Неймана, язык программирования фон Неймановского типа допускает согласованное использование различных типов данных в вычислениях. Выражения со смешанными типами явно (с помощью функций согласования типов) или неявно (функции преобразования осуществляются компилятором или языковой средой по умолчанию) приобретают вполне определённый однозначный смысл.

Помимо соответствия литерного и целого типов, заслуживает внимания согласование целого и вещественных типов, как близких с математической точки зрения (числовых), хотя весьма различных по внутримашинному представлению значений. Оно может осуществляться как округлением (`round()`), так и отбрасыванием дробной части (`trunc()`). Обратное преобразование (из целого в вещественный) производится по умолчанию

в случае необходимости приведения смешанного выражения к более сложному вещественному типу. Преобразование целый-вещественный часто реализуется интерпретативно, и, следовательно, достаточно трудоёмко. В современных процессорах это преобразование возлагается на аппаратуру (математический сопроцессор).

За трёхтиповым примером можно обратиться к программе печати графиков Н. Вирта ([9], с. 35–36).

В расширениях языка Паскаль вопрос согласования типов решён более систематически, так, что имя любого типа может использоваться как функция преобразования к нему: `i := integer(r)`. Кроме того, возможна трактовка значения одного типа в любом другом типе с соответствующей сменой интерпретации внутреннего представления (низкоуровневое средство). Это возможно даже в стандартном Паскале (записи с вариантной частью).

3. Двоичный реверс машинного слова на Си

Предположим, на вход подаётся целое число в десятичной системе счисления, целого типа с длиной 32 бита. Тогда реверс двоичного слова будет выглядеть так:

```
#include <stdio.h>
int a, b;
int main()
{
    scanf("%d",&a);
    b=0;
    while (b<32) {
        printf("%d", ((a>>b)&1));
        b++;
    }
    return(0);
}
```

Другой вариант

```
#include <stdio.h>
int main()
{
    int a;
    while(scanf("%d",&a)==1){
        int b,t=1,n=0;
        b=a;
        while(b>1){
            b/=10;
            t*=10;
            n++;
        }
        n++;
        for(int i=0;i<n;i++){
            printf("%d",a%10);
            a/=10;
        }
        printf("\n");
    }
    return 0;
}
```

15 билет

1) эквивалентность диаграмм и программ (стр.54)

Теорема 2.2.6. Каждой программе P , задающей МТ $T = (A, Q, P, q_0)$, можно эффективным образом сопоставить диаграмму D , образованную символами элементарных МТ, так, чтобы МТ, определяемая этой диаграммой, моделировала бы машину T .

▷ Нам нужно указать способ (алгоритм) эффективного построения диаграммы по программе P , а потом убедиться, что для исходной МТ и МТ, определяемой диаграммой D , выполняются все пять условий определения 2.2.5.

2.2.6.1. Построение диаграммы D . Каждой строке $(q, a, v, q') \in P$ поставим в соответствие на диаграмме символы $\cdot v \cdot$. Для $v = s$ ничего ставить не надо, ведь пустые действия на диаграмме не отображаются. В этом случае окончанием работы машины, определяемой диаграммой, будет правая точка предыдущего элемента диаграммы.

Замечание. Если бы наши программы были бы в пятерках, непосредственной трансляции команды в элемент диаграммы здесь бы не получилось. Вот ещё одна причина использования четвёрок!

Поставим еще одну точку (\bullet), соответствующую начальному состоянию диаграммы, и соединим ее стрелками \rightarrow с левыми точками всех символов v , соответствующих строкам вида $(q_0, a_i, v, q) \in P$. Для каждой пары строк $(q, a_k, v, q') \in P$ и $(q', a_j, v', q'') \in P$ соединим стрелкой \rightarrow правую точку символа v , соответствующую строке $(q, a_k, v, q') \in P$, с левой точкой символа v' , соответствующей строке $(q', a_j, v', q'') \in P$ и надпишем над стрелкой букву a_j . В полученной таким образом диаграмме произведем необходимые упрощения, описанные в конце п. 2.2.3. Диаграмма D построена.

2.2.6.2. Доказательство того, что машина T_D , определяемая диаграммой D , моделирует машину T . Для этого достаточно показать, что выполнены условия (1)–(5) определения 2.2.5. Условие (1) выполняется потому, что алфавиты машин T и T_D совпадают.

Состояниям машины T_D соответствуют точки на диаграмме D . По построению диаграммы каждому состоянию q машины T соответствует одна или две точки на диаграмме

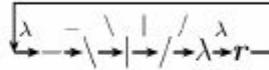
D. В последнем случае сопоставим состоянию q правую точку соответствующего символа на диаграмме. Таким образом, условие (2) выполнено.

Условия (3)–(5) выполнены, так как диаграмма D определяет ту же последовательность элементарных действий, что и программа P , за исключением того, что у машины T_D могут оказаться лишние «пустые» действия, соответствующие переходам от левых точек символов v к правым. \square

Замечание. Процедура построения диаграммы МТ по ее программе (последовательности четверок) является алгоритмом и, следовательно, может быть автоматизирована. В соответствии с этим алгоритмом может быть построен транслятор программ Тьюринга в диаграммы, причем основная сложность такой трансляции заключается в генерации диаграммы, представление в ЭВМ которой одновременно должно быть удобным и для визуализации, и для обработки, и, конечно же, для выполнения. В современных системах автоматизации программирования (CASE) диаграммы являются одним из самых распространенных способов представления алгоритмов, автоматически преобразуемых в программы (иногда наоборот!). Как было сказано выше, именно диаграммный способ построения является наиболее удобным для нисходящей разработки алгоритма.

Пример 2.2.7. Рассмотрим диаграмму, полученную в результате применения теоремы к программе 2.1:

После преобразований эта диаграмма примет вид:



Теорема 2.2.8. Каждой диаграмме Тьюринга D может быть *эффективным образом* сопоставлена программа P так, что МТ, описываемая этой программой, моделирует МТ, описываемую диаграммой D .

▷ Доказательство снова состоит из двух этапов.

2.2.8.1. Построение программы P . Заменим на диаграмме D все символы неэлементарных МТ их диаграммами, продолжая такую замену до тех пор, пока на диаграмме не останутся только обозначения элементарных МТ. В полученной диаграмме перенумеруем одинаковые символы элементарных МТ, снабдив их числовыми индексами. Перенумеруем также все точки, соответствующие *выходным* состояниям. Каждому символу элементарной МТ сопоставим программу этой машины из п. 2.2.3, пометив каждое состояние этой программы дополнительными индексами j и v , где v совпадает с названием соответствующей элементарной МТ, а j — номер, присвоенный этой элементарной МТ. Например, элементарной машине r_j (j -е вхождение элементарной МТ r в диаграмме) будет сопоставлена программа

$$\begin{array}{ll} (q_0^{v,j}, \lambda, r, q_1^{v,j}) & (q_1^{v,j}, \lambda, s, q_1^{v,j}) \\ (q_0^{v,j}, a_1, r, q_1^{v,j}) & (q_1^{v,j}, a_1, s, q_1^{v,j}) \\ \dots & \dots \\ (q_0^{v,j}, a_p, r, q_1^{v,j}) & (q_1^{v,j}, a_p, s, q_1^{v,j}) \end{array}$$

Точке с индексом j сопоставим программу вида

$$\begin{array}{l} (q_j, \lambda, s, q_j) \\ (q_j, a_1, s, q_j) \\ \dots \end{array}$$

$$(q_j, a_p, s, q_j)$$

Соединим все программы, сопоставленные символам элементарных МТ и точкам, в одну и произведем в полученной программе следующие изменения:

- если на диаграмме символы элементарных МТ v_j и v_k соединены стрелкой \xrightarrow{a} , то строку $(q_1^{v,j}, a, s, q_1^{v,j})$ заменим на $(q_1^{v,j}, a, a, q_0^{v,k})$, т. е. останов машины v_j заменим на перезапись буквы рабочей ячейки с последующим переходом прямо в начальное состояние машины v_k , минуя конечное состояние машины v_j ;
- если на диаграмме между символами элементарных МТ v_j и v_k стоит точка, то проведем вышеуказанную замену для всех строк вида (q_i, a, s, q_i) , тем самым предотвращается остановка машины на промежуточных этапах, изображаемых точками — «пустыми» машинами;
- введем новое начальное состояние q_0 для генерируемой программы (оно соответствует крайней левой точке диаграммы) и добавим к программе пустые командные строки $(q_0, \lambda, \lambda, q_0), (q_0, a_i, a_i, q_0)$ ($i = 1, 2, \dots, p$), где a_i — символы элементарных МТ, непосредственно следующих за крайней левой точкой диаграммы. Эти строки являются модифицированной программой элементарной машины «●», в которой остановка заменена перезаписью буквы в рабочей ячейке, т. е. реализуют холостой ход конструируемой программы;
- перенумеруем все состояния программы, за исключением нового q_0 произвольным образом, введя сплошную нумерацию состояний (например, в лексикографическом возрастании индексов: $q_i^{v,j} \rightarrow q_k$. Постройте сами такое отображение индексов по схеме $\{l, r, s, \lambda, a_i\} \times \mathbb{N}^2 \rightarrow \mathbb{N}$).

На этом построение программы P завершается.

2.2.8.2. Проверка условий (1)–(5) определения 2.2.5

Условие (1) выполняется, так как рабочие алфавиты рассматриваемых МТ совпадают. Условие (2) выполняется по построению программы P . Условия (3)–(5) выполняются потому, что обе МТ выполняют над сообщением, записанным на ленте (по построению программы P) одни и те же элементарные действия в одной и той же последовательности (дополнительные элементарные действия имеют вид (q, a, a, q') , т. е. не меняют ситуации на ленте). \square

В заключение теоремы можно сказать, что трансляция любой высокоуровневой визуальной диаграммы в низкоуровневый машинный код описана нами достаточно конструктивным алгоритмическим образом. Здесь, как и раньше, основные трудности реализации заключаются в эффективном представлении диаграмм в памяти ЭВМ. Такого рода трансляторы диаграмм в программный код имеются в CASE-системах автоматизации программирования, т. н. *диаграммерах*, выходными языками которых обычно являются C++, HTML, JavaScript или SQL. Кроме того, необходимо заметить, что доказательство взаимной эквивалентности программ и диаграмм Тьюринга существенно проще, чем, например, строгое сопоставление Паскаля и Си.

2) Тип литерный(стр.172)

Константы литерного типа — самоопределённые термы — заключаются в апострофы или кавычки: '3', 'a'. Множество значений определяется конкретной кодировкой: ASCII, КОИ-8 и т.д. Эта кодировка задаёт порядковый номер литеры, определяемой функцией преобразования типа `ord`, связывающей литерный тип с поддиапазоном целого (обычно [0..255]), и предопределяет упорядоченность множества значений так, что имеет смысл понятия следующей и предыдущей литеры (`succ` и `pred` соответственно).

Одновременно кодовая таблица задаёт и обратное соответствие литер и внутренних кодов (в Паскале это функция `chr`). Таким образом, для всякой литеры "c" имеет место соотношение `c = chr(ord(c))` и наоборот (для малых положительных целых, обычно в диапазоне [0..255]) `i = ord(chr(i))`. Кроме того, `c = succ(pred(c))`

= pred(succ(c)). Для языка Си все условности отброшены и транслятор осуществляет неявное автоматическое преобразование int - char: i = с или с = i. Тип литерный имеет минимальный набор операций и отношений, включающий в себя присваивание : = и стандартный набор отношений. Литерный тип используется для ввода-вывода и обработки текстовых данных (в том числе и для изображений значений других типов словами – цепочками знаков). По этой причине в языке Паскаль литерный тип является «более элементарным», чем другие типы. Основные входной и выходной файлы Паскаль-программы (input и output) – литературного типа. Имеется соответствующий предопределенный файловый тип text. В настоящее время ввод-вывод утрачивает свою литературную ориентацию, приобретая графический и мультимедийный характер. Однако литературный тип весьма важен как простой и удобный стандарт.

3) проверка на палиндромию машинного слова на си

Эта надёжнее:

```
#include <stdio.h>
```

```
int r_offset(int a, int offset) {
    return a >> offset;
}

int main() {
    int a = 0x80000000;
    int b;
    int counter = 0;
    scanf("%d", &b);
    while ((r_offset(a,counter) & b) == 0) {
        counter++;
    }

    int len = 32 - counter;
    int flag = 0;
    for (int i = 0; i < len / 2; i++) {
        if ((b >> (len - 1 - i) & 1) != (b >> i & 1)) flag = 1;
    }
    if (flag) printf("NOT PALINDROM");
    else printf("PALINDROM");

    return 0;
}
```

Написано из предположения, что на вход поступает целое число типа int.

```
#include <stdio.h>
int a, b, c;
int main()
{
    scanf("%d",&a);
    b=0;
    c=1;
    while (b<16){
        if (((a>>b)&1)!=(a>>(31-b)&1)) c=0;
        b++;
    }
    if (c==0) printf ("NOT PALINDROM\n");
    if (c==1) printf ("PALINDROM\n");
```

```
return 0;
```

```
}
```

16 билет

1) обработка данных(информации)(стр.30)

Множество сообщений N представляет интерес только тогда, когда ему соответствует (по крайней мере одно) множество сведений I и определено соответствующее правило интерпретации $\varphi: N \rightarrow I$ (см. п. 1.2). Так как множеству сообщений N' тоже соответствует некоторое множество сведений I' (и правило интерпретации φ'), то любое правило обработки сообщений $\nu: N \rightarrow N'$ (см. п. 1.6) приводит к следующей диаграмме:

$$\begin{array}{ccc} N & \xrightarrow{\varphi} & I \\ \nu \downarrow & & \downarrow \sigma \\ N' & \xrightarrow{\varphi'} & I' \end{array} \quad (*)$$

Эта диаграмма определяет соответствие между множествами I и I' . Так как согласно диаграмме $(*)$ каждому сообщению $n \in N$ соответствует пара сведений $i = \varphi(n) \in I$ и $i' = \varphi(\nu(n)) \in I'$, построенное соответствие между I и I' (обозначим его через σ), вообще говоря, не является отображением. В самом деле, если правило интерпретации φ не является однозначным (инъективным, когда разные переходят в разные), т. е. если существуют два различных сообщения $n_1, n_2 \in N$, $n_1 \neq n_2$, передающих одинаковую информацию $i = \varphi(n_1) = \varphi(n_2)$, то может оказаться, что $\varphi'(\nu(n_1)) \neq \varphi'(\nu(n_2))$ и, следовательно, одной информации $i \in I$ будут соответствовать (по крайней мере) две различных информации $i'_1 = \varphi'(\nu(n_1))$ и $i'_2 = \varphi'(\nu(n_2))$.

Если отображение φ обратимо, т. е. если существует отображение φ^{-1} (для нас достаточно, чтобы φ было инъективным отображением), то можно построить отображение σ , определяющее обработку информации $\sigma: I \rightarrow I'$ в виде $\sigma = \varphi' \circ \nu \circ \varphi^{-1}$ так, что $i' = \varphi'(\nu(\varphi^{-1}(i)))$.

Во всех случаях, когда соответствие σ является отображением, правило обработки сообщений ν называется **сохраняющим информацию**. Если правило обработки сообщений ν сохраняет информацию, то диаграмма

$$\begin{array}{ccc} N & \xrightarrow{\varphi} & I \\ \nu \downarrow & & \downarrow \sigma \\ N' & \xrightarrow{\varphi'} & I' \end{array} \quad (**)$$

коммутативна: $\nu \circ \varphi' = \varphi \circ \sigma$. Отображение σ называется в этом случае **правилом обработки информации**.

Обычно обработку информации сводят к обработке сообщений, т. е., исходя из требуемого правила обработки информации σ , пытаются определить отображения ν , φ и φ' таким образом, чтобы диаграмма $(**)$ была коммутативной.

Если σ — обратимое (взаимно однозначное) отображение, т. е. если информация при обработке по правилу σ не теряется, то соответствующую обработку сообщений ν называют **перешифровкой**.

Пусть ν — обратимая перешифровка. Тогда по сообщению $n' = \nu(n)$ можно восстановить не только исходную информацию, но и само исходное сообщение n . Иными словами, в этом случае n' **кодирует** n (см. п. 1.4). Обратимая перешифровка ν называется **перекодировкой**.

Пусть перешифровка ν не является обратимой, т. е. пусть несколько сообщений из N кодируются одним и тем же сообщением из N' . Но так как при перешифровке информация

не теряется, это означает, что исходное множество сообщений N является избыточным: некоторые сообщения из N содержат одну и ту же информацию (дублируют друг друга). В N' таких дублирующих сообщений меньше, чем в N , так как при обработке по правилу ν некоторые из дублирующих друг друга сообщений «сливаются» в одно сообщение. Перешифровка ν , которая не является обратимой, называется **сжимающей**. Сжатию подвергается множество сообщений. То есть в результате необратимой перешифровки сообщений их количество уменьшается, а информация может либо сохраняться, либо теряться.

Пример 1.7.1. Пусть сообщения (a, b) , составленные из пар целых чисел (например, в десятичной позиционной записи), передают информацию «рациональное число r , представленное дробью $\frac{a}{b}$ ». Тогда $N = \mathbb{Z} \times \mathbb{N}$ (где \mathbb{Z} — множество целых чисел, \mathbb{N} — множество натуральных чисел), $I = \mathbb{Q}$ (\mathbb{Q} — множество рациональных чисел). Отображение $\varphi: N \rightarrow I$ не является обратимым, так как при любом целом n парам (a, b) и (na, nb) соответствует одно и то же рациональное число r . Пусть N' — множество пар (p, q) взаимно простых целых чисел и пусть $\nu: N \rightarrow N'$ переводит все (pr, nq) в (p, q) . Тогда ν — сжимающее отображение, а $\varphi': N' \rightarrow I$ — обратимое отображение (мы считаем $I' = I$). Такое отображение ν называется **вполне сжимающей перешифровкой**, поскольку после обработки сообщений соответствие между сообщениями и информацией биективно. Здесь информация не теряется.

Если σ — необратимое отображение, т. е. если разные сведения из I отображаются в одну и ту же информацию $i' \in I'$, то соответствующую обработку сообщений называют **избирательной**. Особенно часто встречается случай, когда $I' \subset I$, а σ — тождественное отображение для всех $i' \in I'$. В этом случае производится **выбор** из данного множества сведений.

Таким образом, «обработка информации» — это, как правило, сокращение количества информации. Во всяком случае, верно утверждение: обработка информации никогда **не добавляет** информацию, она состоит в том, что **извлекает** интересную информацию из той, которая содержится в сообщении.

2) теорема Бёма-Якопини-Миллса

Теорема 2.7.3. (Бёма-Якопини-Миллса) Для любой машины Тьюринга T можно эффективно построить машину Тьюринга S , которая является структурной (т. е. диаграмма которой является схемой) и которая моделирует машину T .

Доказательство этой теоремы состоит в преобразовании каждой части диаграммы машины T в одну из трех структур, приведенных в определении 2.7.2. Каждое такое преобразование (мы будем называть его структурирующим преобразованием) уменьшает неструктурную часть диаграммы. После достаточного количества структурирующих преобразований неструктурная часть диаграммы исчезает. Возможность проведения структурирующих преобразований вытекает из первой теоремы Шеннона, так как каждое такое преобразование сводится к уменьшению числа состояний моделируемой МТ. При этом, естественно, рабочий алфавит моделируемой МТ расширяется (в него вводятся дополнительные буквы). Полного доказательства теоремы 2.7.3 мы приводить не будем из-за его громоздкости.

Теорема 2.7.3 была сформулирована Бёмом и Якопини в 1966 г., однако, как было впоследствии установлено сотрудником фирмы IBM Миллсом, их доказательство было неполным. Полное доказательство было опубликовано Миллсом в 1972 году [42, 38].

Ниже следует полученное Д. В. Рисенбергом чисто тьюринговское, диаграммное доказательство теоремы, основанное на идеях Клода Шеннона о сокращении числа состояний произвольной машины Тьюринга за счет увеличения числа букв рабочего алфавита.

2.7.3.1 Доказательство

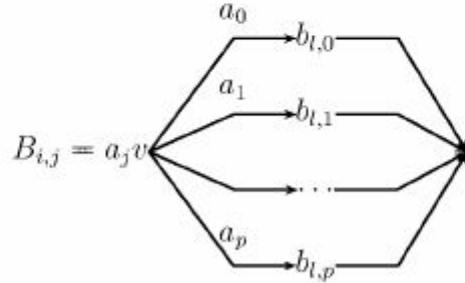
Пусть машина T имеет алфавит $A = \{a_1, \dots, a_p\}$ и набор состояний $Q = \{q_0, \dots, q_s\}$. Тогда алфавит машины T' будет содержать помимо алфавита A еще $(p+1)(s+1)$ знаков $b_{i,j}$.

Каждой команде из программы машины T' , не являющейся терминальной (q_i, a_j, s, q_l) , поставим в соответствие некоторую структурную машину B_{ij} по следующему правилу:

- Если команда машины T имеет вид (q_i, a_j, a_k, q_l) , то ей соответствует машина перезаписи буквы

$$B_{i,j} = b_{l,k}.$$

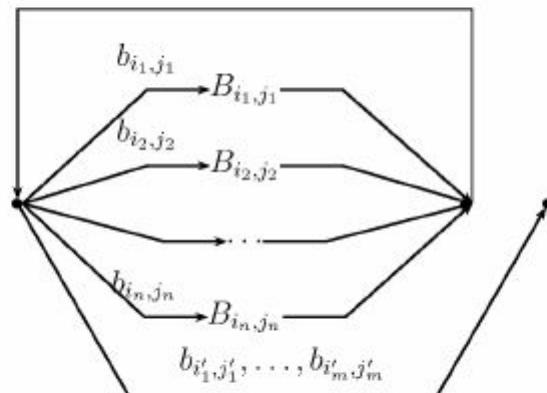
- Если команда машины T имеет вид (q_i, a_j, v, q_l) , $v \in \{l, r\}$, то ей соответствует машина движения



- Если команда машины T имеет вид (q_i, a_j, s, q_l) ($i \neq l$), то ей соответствует стоп-машина

Таким образом, каждый знак $b_{i,j}$ соответствует определяющей паре (q_i, a_j) исходной машины T и неявно содержит в себе информацию о номере состояния, в котором она находится. Таким образом, сгенерирована матрица инциденций B_{ij} , отражающая наличие связей всех состояний (точкомест) в диаграмме.

Теперь можно построить диаграмму машины T' :



Здесь введена сплошная (от 1 до n) нумерация для пар (i, j) таких, что пара (q_i, a_j) определяет некоторую команду машины T , не являющуюся терминальной. Аналогично была введена сплошная (от 1 до m) нумерация для пар (i, j) , для которых пара (q_i, a_j) определяет терминальную команду машины T . Сам порядок, в котором перенумерованы пары, не важен.

Исходя из построенной диаграммы, можно заметить, что машина T' является структурной, т. к. структурными являются все машины $B_{i,j}$.

Осталось показать, что машина T' моделирует машину T .

Конфигурации

$$[\lambda a_{l_1} \dots a_{l_k} \dots a_{l_n} \lambda] \\ q_i$$

машины T поставим в соответствие конфигурацию

$$[\lambda a_{l_1} \dots b_{i,l_k} \dots a_{l_n} \lambda] \\ q^*$$

машины T' , где состояние q^* может быть любым. Тогда начальной конфигурацией машины T' будет

$$[\lambda a_{l_1} \dots b_{i_{\text{нач}}, l_{k_{\text{нач}}} \dots a_{l_n} \lambda], \\ q$$

соответствующая начальной конфигурации машины T (здесь q — начальное состояние T').

Рассмотрим выполнение нетерминальной команды (q_i, a_j, v, q_l) , где $v \in A \cup \{\lambda\} \cup \{l, r, s\}$, а $i \neq l$.

Если $v \in A \cup \{\lambda\}$, то машина T' просто заменяет в своей рабочей ячейке знак $b_{i,j}$ на $b_{l,k}$. Если $v \in \{l, r\}$, то машина машина T' сначала считает из ячейки букву $b_{i,j}$, восстановит в ней букву a_j , сдвигается в направлении v и в новой рабочей ячейке записывает букву $b_{l,j'}$, если до этого в ней была записана буква $a_{j'}$. Если $v = s$, то машина T' заменяет в рабочей ячейке букву $b_{i,j}$ на $b_{l,j}$. Наконец, если команда (q_i, a_j, v, q_l) является терминальной ($l = i, v = s$), то машина T' сразу же выходит из цикла и завершает работу.

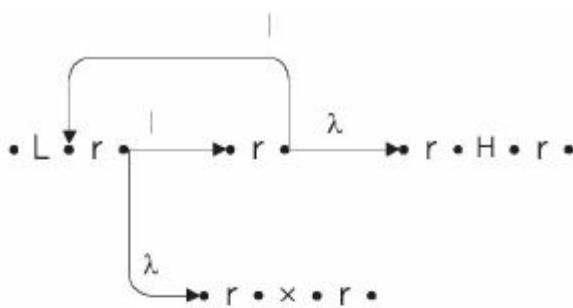
Таким образом, теорема Бойма-Якопини-Миллса доказана.

Замечание. Данное доказательство неконструктивно, но весьма миниатюрно и изящно как будто бы его делал не программист, а математик. Харлан Миллс, популяризируя результаты Бойма и Якопини для программистов-практиков из IBM, дал конструктивное доказательство теоремы на базе блок-схем программ.

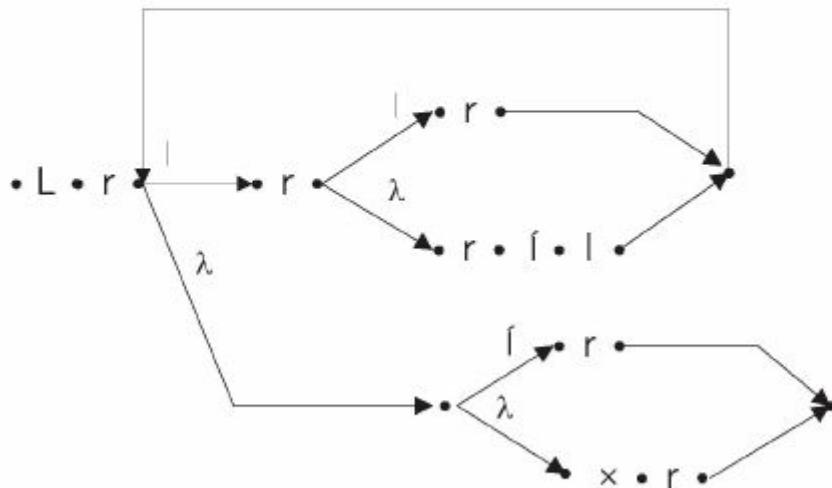
Из теоремы Бойма-Якопини-Миллса вытекает важное

Следствие. Всякая диаграмма может быть преобразована к виду, не использующему произвольные передачи управления — стрелки, соединяющие элементы диаграммы неструктурированными способами. В практике программирования это означает, всякая программа может быть преобразована в эквивалентную, не содержащую ни меток, ни операторов безусловного перехода **goto**. Программирование без **goto** вместе с исходящей разработкой было одним из элементов технологии структурного программирования — одной из первых технологий (после процедурного и модульного программирования).

Пример 2.7.4. Рассмотрим пример структурированного преобразования диаграммы Тьюринга. Машина Тьюринга, просматривающая слово $w \in \{\}\ast$ и печатающая в качестве результата своей работы букву Ч, если слово w содержит четное число палочек, и букву Н, если w содержит нечетное число палочек, описывается следующей неструктурированной диаграммой:



Эта диаграмма может быть заменена следующей схемой:



В рассмотренном примере удалось структурировать диаграмму МТ, не вводя вспомогательных букв в ее рабочий алфавит (роль такой буквы удалось возложить на Н).

В середине XX века многими отечественными исследователями были предложены свои исчисления схем программ [39].

3) написать НАМ переводящий число из двоичной системы счисления в восьмеричную

Решение:

```

a1->a
a0->0a
a->b
000b->b0
001b->b1
010b->b2
011b->b3
100b->b4
101b->b5
110b->b6
111b->b7
10b->b2
11b->b3
01b->b1
1b->b1
0b->b
00b->b
b->.
->a
  
```

17 билет

1) Конструктивное описание процесса обработки дискретных сообщений

(см. билет 13)

2) Тип целый (стр.146)

(integer, long int и пр). Значениями машинного целого типа являются все математические целые числа, заключенные между двумя фиксированными значениями Min и Max, являющимися атрибутами конкретного целого типа (разные границы у integer и long long). Формульно: $Z = \{z \in Z | MIN \leq z \leq MAX\} \cup \{\text{Null}, \text{предопределенное значение}\}$

Над такими типами можно проводить следующие операции: +, -, *, /, %.

Некоторые свойства операций и отношений:

1. коммутативность сложения и умножения: $UX, K 6 Z X + Y = Y + X, X * Y = Y * X;$

2. если $X > Y > 0$ или $X < Y < 0$, то $(X - Y) + Y = X;$

3. монотонность операций: $VX, Y \in Z$ из $0 < X < A$ и $0 < Y < B$ следует $X + Y < A + B, X - B < A - Y, X * Y < A * B, X + B < A + Y$; Всё это верно только если $A + B < M A X$ и $A * B < M A X$ соответственно.

4. если хотя бы один из операндов операции целого типа имеет значение _L или T, то результат операции тоже имеет значение _L или T соответственно. Если один операнд двуместной операции имеет значение _L, а другой T, то результат операции имеет значение _L;

5. если один из операндов отношения целого типа имеет значение _L или T, то результат имеет значение _L. Строго говоря, это неопределенное значение принадлежит логическому типу.

6. Если один из операндов Null или inf (переполнение), то результат будет тоже Null или inf соответственно

3) Составить ДТ по сложению двоичных чисел

состояния машины

00, ,<,01

01,0,1,02

02,1,<,01

01,1,0,03

03,0,<,03

03,1,<,03

03, ,<,04

04,0,1,05

05,0,>,05

05,1,>,05

05, ,>,00

00,1,>,00

00,0,>,00

04,1,0,02

02,0,<,04

04, ,1,05

01, ,>,06

06,1,>,06

06, ,<,07

07,1, ,06

07, ,#07

Билет 18 (вроде)

1) Необходимость формального определения алгоритма

см. билет 1

2) Вызов процедур и функций. Передача аргументов(стр.186)

Особенностью описания подпрограмм является наличие в заголовке так называемых **формальных параметров**, *параметризующих* тело программы. Этим достигается универсальность подпрограммы по сравнению с блоком. Формальные параметры являются локальными объектами подпрограммы и поэтому они по существу лишь обозначают объекты подпрограммы и им не отводится памяти при трансляции. Подобно всяким локальным объектам, память для формальных параметров выделяется только при входе в блок подпрограммы.

При вызове подпрограммы на место формальных параметров подставляются фактические параметры, так что их значения или ссылки на них замещают формальные параметры в период данного вызова подпрограммы. При другом вызове на место этих же формальных параметров могут подставляться другие фактические параметры, что и обеспечивает динамизм и универсальность подпрограмм.

В отличие от процедур подпрограммы-функции имеют один явный результат, как правило скалярный, явно присваиваемый имени функции. Обычно этот результат оставляется на определенном регистре соответствующего регистрового файла. Нескалярный параметр заменяется ссылкой на него, которая всегда скалярна; в Си и расширениях Паскаля тип результата функции может быть структурным. Кроме того, признаком хорошего тона является использование всех параметров функции как констант, так чтобы результат функции был ее единственным эффектом. Обращение к функции происходит, как правило, без специальной инструкции вызова в выражении соответствующего типа. В соответствии с парадигмой императивных языков фон Неймановского типа вычисление выражения понуждает к выполнению все указуемые им функции. Указатель функции, т. е. имя функции с конкретным набором параметров, является таким же элементом выражения, как константа, переменная, элемент массива.

3.4.3 Передача параметров

Передача параметров является тем механизмом, с помощью которого устанавливается связь по данным между подпрограммой и внешней средой, из которой происходит обращение к подпрограмме.

В современных языках программирования существует несколько способов передачи параметров, различающихся степенью и характером взаимосвязи по данным вызывающей и вызываемой подпрограмм. Эти способы разбиваются на четыре группы по характерным особенностям передачи, хотя в конкретных языках эти качества могут комбинироваться или отсутствовать в чистом виде.

Тетрадный лист с видами

3) ДТ: перевод числа из двоичной системы в натуральную

19 билет

1. Кодирование(стр.19)

Кодом называется правило, описывающее отображение $C: A \rightarrow A'$ алфавита (набора знаков) A на другой набор знаков A' . Здесь «на» означает сюръекцию, отображение, при котором каждый элемент из A' имеет прообраз из A . Если при этом алфавит A' содержит меньше знаков, чем A , то образами некоторых знаков из A будут комбинации (конечные последовательности) знаков из A' (слова над алфавитом A'), т. е. знаки более высокого уровня. Например, отображение восьмеричных цифр в двоичные триады дает двоичнокодированную запись восьмеричного числа ($0_8 \rightarrow 000_2, 1_8 \rightarrow 001_2, \dots, 7_8 \rightarrow 111_2$).

В отличие от тайнописи, когда скрывается сам факт существования сообщения, при кодировании меняются лишь изображения букв (знаки). Смысл знаков сохраняется. Следовательно, при кодировании сообщения n получается новое сообщение $C(n)$, которое

содержит ту же информацию, что и n .

При этом изменяется правило интерпретации сообщений. Если до кодирования для интерпретации использовалось отображение φ_0 , то после кодирования сообщения будут интерпретироваться с помощью отображения $\varphi_1 = C^{-1} \circ \varphi_0$, где C^{-1} — отображение, обратное отображению C (отображение **декодирования**). Следовательно, отображение C должно допускать существование обратного отображения, т. е. быть взаимно однозначным (биективным). Если известно отображение C^{-1} (или C), или известен способ построения отображения C^{-1} (или C), то говорят, что известен **ключ** кода C .

В качестве примера кодов рассмотрим семейство кодов Юлия Цезаря. В этих кодах знаки алфавита A , содержащего m знаков, заменяются другими знаками того же алфавита по следующему правилу: выбирается постоянное целое число $p \in [0, m]$ и k -му знаку алфавита A ставится в соответствие его $(k+p)$ -й знак, если $k+p \leq m$, и $(k+p-m)$ -й знак, если $k+p > m$. Если, например, A — русский алфавит и $p = 3$, то слово *информатика* будет закодировано словом *лрчсупххнг*. Легко видеть, что ключом для кода Цезаря является число p , определяющее величину сдвига алфавита A при кодировании. Другими примерами являются азбука Морзе и различные кодировки текстовых данных в ЭВМ (КОИ-8, ASCII и т. п.).

Продолжим рассмотрение различных аспектов кодирования. Кодирование, при котором каждый *образ* является отдельным знаком, называется **шифрованием**. Традиционный шифр — это книга, в которой словам естественного языка сопоставлены группы цифр или букв. Видоизменяя сообщение, шифровка скрывает от непосвященного его смысл.

В рассмотренных ранее (симметричных) системах шифрования операции кодирования и декодирования являются взаимно обратными функциями, т. е. зная способ шифрования можно, действуя наоборот, расшифровать сообщение. Современная теория чисел дает способ создать систему шифрования, в которой используются два ключа — один для кодирования, другой для декодирования (т. н. **ассимметричное шифрование**). Система построена таким образом, чтобы, зная шифрованное сообщение и способ кодирования, было невозможно расшифровать сообщение за приемлемое время (подробнее см. [77]). Современная криптография основана на гипотезе об отсутствии каких-либо закономерностей в распределении простых чисел. В 2004 г. эта гипотеза была опровергнута, опубликовано доказательство, поставившее под сомнение всю просточисленную криптографию.

Одностороннее шифрование используется для хранения паролей (например, в Unix), и строго говоря кодированием не является. В этом случае используется неоднозначная (например, периодическая) функция. Для каждого шифрованного пароля получается множество вариантов дешифрованных, что затрудняет проникновение в систему.

После того, как кодирование и шифрование достаточно проиллюстрированы, можно избавиться от ненужного разнообразия алфавитов. Иногда оказывается удобным кодировать знаки рассматриваемого алфавита A в начальном отрезке счетного множества стандартных знаков $\omega = b_1, b_2, \dots, b_m, \dots$. При этом алфавиту A , содержащему m знаков, ставится в соответствие стандартный m -значный алфавит $\omega = b_1, b_2, \dots, b_m$, так, что первая буква алфавита A кодируется стандартным знаком b_1 , вторая буква — знаком b_2 , и т. д. Если, например, A — русский алфавит, то ему соответствует стандартный алфавит ω , в котором слово **информатика** записывается в виде $b_{10}b_{15}b_{22}b_{16}b_{18}b_{14}b_1b_{20}b_{10}b_{12}b_1$.

Наряду со стандартным набором знаков ω мы будем использовать расширенные стандартные наборы знаков $\omega_0, \omega_{-1}, \dots$, которые получаются добавлением к набору ω несобственных знаков b_0, b_{-1}, \dots , которые используются для формирования разделителей, в частности, $\omega_0 = \omega + b_0$. Несобственный знак b_0 является обозначением для пробела (' ', λ).

(1.4.1) Утверждение. Произвольный конечный набор знаков (алфавит) может быть закодирован знаками набора $\omega = (b_0, b_1)$.

► Укажем способ кодирования. Знаки исходного набора могут быть закодированы знаками стандартного алфавита ω_m , где m — число знаков в исходном наборе. Пусть $b_i \in \omega_m$. Сопоставим ему код $b_0b_1\dots b_1$ ($i+1$ раз b_1) (последний код мы будем для краткости обозначать $b_0b_1^{i+1}$). Утверждение доказано. □

Иногда для большей наглядности будут использоваться другие обозначения знаков набора ω : b_0 будет обозначаться знаком λ , а b_1 — знаком | («палочка»).

2. Обобщенная инструкция цикла(стр.135)

Пусть требуется вычислить значение $f(n) = n!$ для заданного значения n . Поставим целью составить программу вычисления $n!$ для любого задаваемого значения n . Очевидно, что

это можно сделать по следующим соотношениям:

$$\begin{aligned} &\text{при } n = 0 \quad f(n) = 1; \\ &\text{при } i > 0 \quad (i = 1, 2, \dots, n) \quad f(i) = f(i - 1) * i, \end{aligned}$$

дающим правила вычисления каждого последующего значения функции через предыдущее. Следовательно, программа должна содержать: определение начального значения функции $f(0)$ и завершение вычисления, если $n = 0$; последовательное умножение значения функции на $i = 1, 2, \dots, n$ до тех пор, пока $i \leq n$, где n — задаваемое значение.

Из сказанного следует, что есть необходимость в специальном программном средстве для организации многократного выполнения умножения текущего значения искомой функции на значение аргумента, увеличившееся на единицу. Повторяться эти вычисления должны до тех пор, пока значение аргумента не превысит некоторого заранее заданного значения.

Такой инструкцией является инструкция цикла, которую мы определим для общего случая следующим образом:

```
DO <предохранитель1> <охраняемая инструкция1>
  ┌<предохранитель2> <охраняемая инструкция2>
  ...
  ┌<предохранительm> <охраняемая инструкцияm>
OD
```

Несмотря на то, что инструкция цикла **DO-OD** внешне похожа на инструкцию ветвления, правила выполнения этой инструкции определяются иначе:

- Если среди предохранителей инструкции **DO-OD** *один и только один* принимает значение **И**, то соответствующая охраняемая инструкция выполняется, и после этого вновь осуществляется *одновременная и независимая проверка всех предохранителей* инструкций цикла. Следует обратить внимание на то, что охраняемой инструкцией может быть композиция или ветвление.
- Количество предохранителей, принимающих значение **И**, также *может быть более одного*. В этом случае не предполагается, что охраняемые инструкции упорядочены и выбор инструкции для выполнения очередного повторения цикла осуществляется вне связи с порядком их написания в инструкции **DO-OD**, т. е. недетерминированным образом.
- Если среди предохранителей инструкции **DO-OD** нет *ни одного*, принимающего значение **И**, то, *в отличие от ветвления*, выполнение инструкции заканчивается естественным образом в связи с отсутствием инструкции, открытой для продолжения работы цикла.
- Выполнение каждой охраняемой инструкции должно приводить к изменению аргументов предохранителей. В противном случае выполнение инструкции **DO-OD** может никогда не закончиться.

3. Написать НАМ построения машинного слова по заданной двоичной маске

Дана маска и число, и нужно сделать конъюнкцию маски и числа(перемножить, и перевести в натулярную систему счисления) (Прим. Прога с моей лабы. Не уверена что она делает именно это, в унарную систему не переводит точно, да и за верность не ручаюсь) при входных 111100\$1001 дает <100>**

```
0*0 -> 00*
1*1 -> 11*
1*0 -> 01*
0*1 -> 10*
00* ->
10* ->
01* -> #0
11* -> #1
0$0 -> $00*
0$1 -> $10*
1$1 -> $11*
1$0 -> $01*
1$# -> $#0#
0$# -> $#
$ -> <>
>0 -> >
>1 -> >
>#0 -> 0>
>#1 -> 1>
> ->
<0 -> <
< ->
```

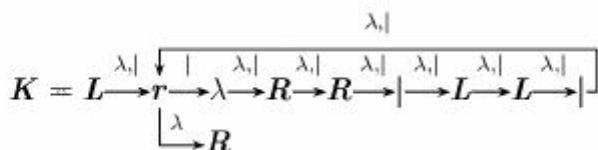
Билет 20

1. Диаграммы Тьюринга(стр.47)

Диаграммы Тьюринга представляют одни МТ через другие, более простые МТ иным, визуально-топологическим способом, причём, как будет показано далее, этот способ не менее строг и полон, нежели "обычные" МТ. Так, машина, копирующая на ленте записанное на ней слово, рассмотренная в примере (2.2.4), может быть представлена через МТ, которые ищут начало слова на ленте, конец слова на ленте, копируют одну из букв слова и т. д. (имена состояний подобраны в этом примере так, чтобы легко было выделить более простые МТ). Эти более простые МТ в свою очередь могут быть представлены через еще более простые МТ (это тоже нетрудно проиллюстрировать на примере (2.2.4)) и т. д. Такой нисходящий процесс представления МТ через более простые МТ должен обязательно оборваться, так как рано или поздно мы сведем описание каждой из рассматриваемых МТ к элементарным действиям, введенным при определении МТ. При этом рассматриваемая МТ будет описана через **элементарные** МТ, т. е. такие, которые уже нельзя описать через более простые МТ, так как каждая из них выполняет всего одно элементарное действие и останавливается.

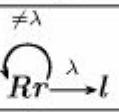
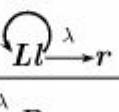
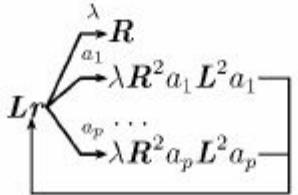
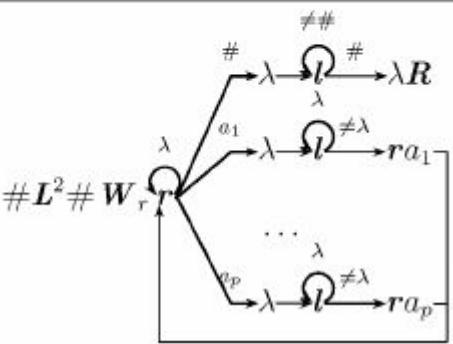
Элементарные МТ над алфавитом $A_p = (a_1, a_2, \dots, a_p)$ определяются следующими программами.

Таким образом, диаграмма Тьюринга состоит из символов (имен) машин Тьюринга, точек и стрелок, над которыми надписаны знаки алфавита A_p . В качестве примера рассмотрим диаграмму МТ из примера (2.2.4). Она имеет вид:



Приведем некоторые примеры диаграмм Тьюринга [7]:

Символ	Действие	Диаграмма
I	$[\lambda w(\lambda)\lambda] \xrightarrow{*} [\lambda w\lambda w^{-1}(\lambda)\lambda]$	
K_n	$[\lambda w_1 \lambda w_2 \lambda \dots \lambda w_n(\lambda)\lambda] \xrightarrow{*}$ $\Rightarrow [\lambda w_1 \lambda w_2 \lambda \dots \lambda w_n \lambda w_1(\lambda)\lambda]$	

Символ	Действие	Диаграмма
R	$[(\lambda)w\lambda] \xrightarrow{*} [\lambda w(\lambda)\lambda]$	
L	$[\lambda w(\lambda)\lambda] \xrightarrow{*} [(\lambda)w\lambda]$	
\mathfrak{R}	$[(\lambda)w_1\lambda w_2\lambda \dots \lambda w_n\lambda] \xrightarrow{*} [\lambda w_1\lambda w_2\lambda \dots \lambda w_n(\lambda)\lambda]$	
\mathfrak{L}	$[\lambda w_1\lambda w_2\lambda \dots \lambda w_n(\lambda)\lambda] \xrightarrow{*} [(\lambda)w_1\lambda w_2\lambda \dots \lambda w_n\lambda]$	
K	$[\lambda w(\lambda)\lambda] \xrightarrow{*} [\lambda w\lambda w(\lambda)\lambda]$	
W_r	$[(\lambda)w\lambda] \xrightarrow{*} [\lambda\lambda \dots (\lambda)]$	
W_l	$[\lambda w(\lambda)] \xrightarrow{*} [(\lambda)\lambda \dots \lambda]$	
V	$[\lambda w_1\lambda w_2(\lambda)\lambda] \xrightarrow{*} [\lambda w_2(\lambda)\lambda]$	

2. Небазовые типы данных(стр.175)

Поскольку предусмотреть в языке программирования встроенных базовых типов на все случаи жизни невозможно, обычно предоставляют возможность более или менее тривиального конструирования новых типов данных на основе базовых.

3.2.7.1 Отрезок (диапазон)

Можно было бы объявить отрезок типа, например, следующим образом:

```
type diap = (0.0 .. 2.0; real; real);
```

Здесь в качестве множества значений берётся отрезок вещественного типа, операции и отношения также заимствуются из вещественного типа.

3.2.7.2 Перечисление

В программе управления уличным движением удобно объявить

```
type StreetLight = ((Red, Yellow, Green) ; := ; integer);
```

3. Выдача байта на си

```
int get_byte(int input, int num){  
    return (input >> (num * 8)) & 255; //num-номер байта input-число  
}
```

21 билет

1) Знаки и Символы(стр.18)

Каждое сообщение передается на каком-либо материальном **носителе сообщения**. Для устных сообщений носителем является воздух. Мы будем изучать сообщения, представленные на **долговременных** носителях (таких, как бумага, магнитная лента и т. п.). Такие сообщения называются **письменными**. Процесс нанесения письменных сообщений на носитель называется **записью** или **письмом**, а процесс их воспроизведения — **чтением**.

Письменные языковые сообщения представляют из себя последовательности знаков. Мы будем различать **атомарные** и **составные** знаки.

Атомарным знаком или **буквой** называется элемент какого-либо упорядоченного конечного непустого множества графически (или каким-либо иным образом) отличимых друг от друга литер (предметов, сущностей, объектов, членов), называемого **алфавитом**.

В качестве примеров атомарных знаков (букв) можно рассмотреть русскую (латинскую) букву А, арабскую цифру 5, знак сложения и т. п. В качестве экзотических можно представить себе алфавиты, элементы которых различимы не зритально, а с помощью иных органов чувств: по запаху, на ощупь или по весу). Понятие алфавита имеет важное практическое значение. Оно определяет множество допустимых знаков для записи входных и выходных сообщений на клавиатуре и принтере соответственно.

Составным знаком или **словом** называется конечная последовательность знаков (неважно, атомарных или составных). Множество слов тоже можно рассматривать как набор знаков (алфавит или *словарь*).

Таким образом, письменное сообщение может рассматриваться как один составной знак, либо как последовательность знаков (простых или составных). Если в состав последовательности знаков входят составные знаки, то необходимо иметь специальный знак, который помещается между соседними составными знаками, отделяя их друг от друга. Этот знак называется **разделителем**. Обычно в качестве разделителя используется знак пробела, который мы будем обозначать λ или пропуском (пустое знакоместо).

Атомарные знаки будем называть **знаками 1-го уровня**. Последовательности знаков 1-го уровня образуют знаки 2-го уровня (слова). Последовательности знаков второго уровня (в их составе разделители) — знаки третьего уровня и т. д. Для записи сообщения, которое содержит знаки k -того уровня необходимо иметь $k - 1$ различных типов знаков-разделителей. В естественных языках текст целиком представляет собой знак достаточно высокого уровня. Так, суть письма запорожских казаков турецкому султану может быть выражена всего лишь одним знаком (кукиш). Для разделения слов (знаков второго уровня) используется пробел, для разделения предложений (знаков третьего уровня) используется точка, для разделения групп предложений (знаков четвертого уровня) — абзац и т. д.

С каждым знаком связывается его **смысл** или **семантика**. Знак (некоторого уровня) вместе с сопоставленной ему семантикой называется **символом** (это слово перегружено значением *знака, буквы, литеры*). Отметим, что разные знаки могут изображать один и тот же символ (знаки \bullet и \times в разных языках изображают один символ умножения) и что один и тот же знак может изображать разные символы (λ является символом для буквы греческого алфавита, символом пробела в теории алгоритмов, символом аргумента в λ -исчислении и т. д.). Таким образом, отношение между символом и знаком аналогично отношению между информацией и сообщением (знак — это сообщение символа).

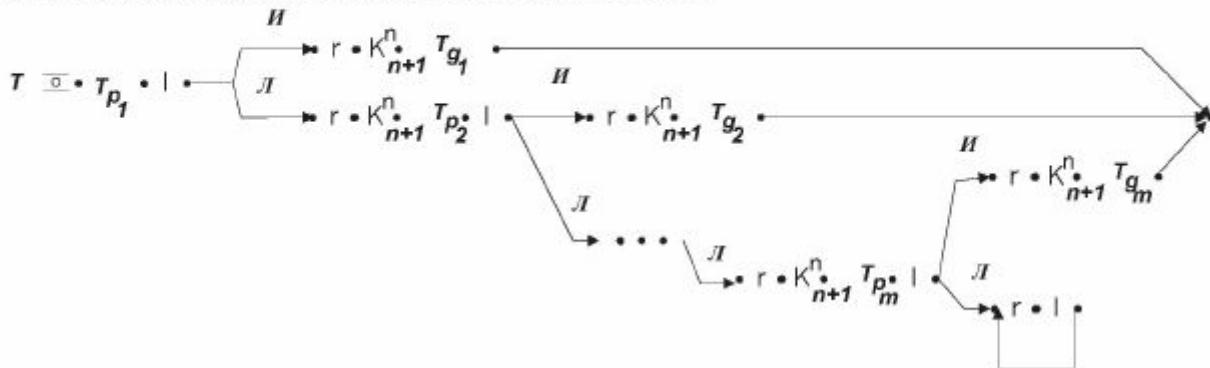
2) Теорема о ветвлении(стр.84)

Теорема 2.6.2. (О ветвлении) Пусть n -местная функция $f : (A_s^*)^n \rightarrow A_t^*$ определяется равенством:

$$f(u_1, u_2, \dots, u_n) = \begin{cases} g_1(u_1, u_2, \dots, u_n), & \text{если } p_1(u_1, u_2, \dots, u_n) = \text{И} \\ g_2(u_1, u_2, \dots, u_n), & \text{если } p_2(u_1, u_2, \dots, u_n) = \text{И} \\ \vdots \\ g_m(u_1, u_2, \dots, u_n), & \text{если } p_m(u_1, u_2, \dots, u_n) = \text{И} \end{cases}$$

где $g_i : (A_s^*)^n \rightarrow A_t^*$ ($i = 1, 2, \dots, m$) — ВТ-функции, а $p_i : (A_s^*)^n \rightarrow \{\text{И}, \text{Л}\}$ ($i = 1, 2, \dots, m$) — ВТ-предикаты. Тогда f является ВТ-функцией, причем МТ, вычисляющая функцию f может быть эффективно построена из МТ, вычисляющих g_i и предикаты p_i ($i = 1, 2, \dots, m$).

▷ Пусть T_{g_i} и T_{p_i} ($i = 1, 2, \dots, m$) — МТ, реализующие нормированное вычисление функций g_i и предикатов p_i ($i = 1, 2, \dots, m$) соответственно (возможность эффективного построения таких МТ для любых ВТ-функций следует из теоремы 2.4.3). Машина T , вычисляющая функцию f , определяется диаграммой:



В случае, когда все p_i ($i = 1, 2, \dots, m$) имеют значение Л, функция f не определена; поэтому в этом случае предусмотрено, чтобы машина T никогда не останавливалась. Проверка того, что машина T действительно вычисляет функцию f , производится аналогично соответствующей проверке, проведенной при доказательстве теоремы 2.5.1. Теорема доказана. \square

Замечание. Данная теорема описывает m -звенное ветвление. Каждая ветвь охраняется своим предикатом. Предполагается, что набор предикатов ветвления p_i ($i = 1, 2, \dots, m$) «полон» и «непротиворечив» : при ветвлении один и только один предикат принимает значение «истина». Считается, что набор предикатов полон так, чтобы ветвление **никогда не отказывало**. В реальности программист может просто запланировать резервную ветку, охраняемую предикатом $p_{m+1} = \neg \bigvee_{i=1}^m p_i$, чтобы дополнить систему предикатов. Непротиворечивость же так просто не достигается. Поэтому в простейших последовательных моделях исполнения предикаты проверяются в процессе написания и ветвление выполняется по первой же открывшейся ветке. Сказанное означает, что к выбору набора предикатов для многозвездного ветвления надо подходить очень и очень основательно. Возможна организация ветвлений на противоречивых наборах предикатов, но она носит недетерминированный и, следовательно, неалгоритмический (в классическом понимании этого слова) характер. В этом случае ветвление идет по одной из открывшихся ветвей, выбираемых недетерминированным и неалгоритмическим образом. Подчеркнем, что даже случайный выбор ветки выполняется по некоторому вероятностному закону и, следовательно, детерминирован и алгоритчен.

3) НАМ реверса слова

Пусть на вход подаётся слово на латинице (Если че, алфавит можно будет увеличить)

```
#A->aX
#B->bX
XA->aX
XB->bX
X->
#aA->.AA
#aB->.AB
#bA->.BA
#bB->.BB
#a->A
#b->B
Bb->bB
Ba->aB
Ab->bA
Aa->aA
->#
```

Билет 22

1. Моделирование машины Тьюринга(стр.53)

Определение 2.2.5. Рассмотрим две МТ $T = (A, Q, P, q_0)$ и $T' = (A', Q', P', q'_0)$. Будем говорить, что машина T' моделирует машину T , и обозначать это $T' \simeq T$, если выполняются следующие условия:

1. Указан способ кодирования знаков (букв) алфавита A знаками (буквами или словами) алфавита $A' C: A \rightarrow A'$;

2. Каждому состоянию $q \in Q$ машины T поставлено в соответствие некоторое состояние $q' \in Q'$ машины T' , т. е. определено отображение $Q \rightarrow Q'$.

Условия (1) и (2) позволяют для каждой конфигурации C машины T составить конфигурацию C' машины T' , являющуюся образом конфигурации C , заменяя каждую букву на ленте машины T ее кодом, а состояние q — состоянием q' ;

3. Если C_0 — начальная конфигурация машины T , то ее образ C'_0 — начальная конфигурация машины T' ;

4. Если машина T из начальной конфигурации C_0 после конечного числа *тактов* (переходов от одной конфигурации к другой) останавливается в конфигурации C_k , то

машина T' из начальной конфигурации C'_0 , являющейся образом C_0 , после конечного числа тактов также останавливается в конфигурации C'_k , являющейся образом C_k ;

5. Если из начальной конфигурации C_0 машина T пробегает (конечную или бесконечную) последовательность конфигураций $C_0, C_1, \dots, C_k, \dots$, то каждая последовательность конфигураций, пробегаемая машиной T' из начальной конфигурации C'_0 (образа конфигурации C_0), содержит в качестве подпоследовательности последовательность конфигураций $C'_0, C'_1, \dots, C'_k, \dots$, где C'_i ($i = 0, 1, \dots, k, \dots$) — образы конфигураций C_i машины T .

Из определения 2.2.5 следует, что если машина T' моделирует машину T , то она описывает тот же самый алгоритм, что и T , но, возможно, проходит при выполнении алгоритма большее число промежуточных конфигураций. Таким образом, понятие моделирования вводит бинарное отношение алгоритмического равенства между машинами Тьюринга. Другие операции отношения над алгоритмами будут введены позднее, по мере изложения Тьюринговской теории алгоритмов. Следует заметить, что равенство алгоритмов даже в таком простом случае — машин Тьюринга — требует более сложного определения, чем равенство строк или чисел.

2. Понятие о структурном типе данных(стр.175)

До сих пор значения типов, даже и изображаемые словами, нами считались **атомарными**, не имеющими структуры, хотя очевидно, что во многих задачах удобно иметь дело со значениями, имеющими ту или иную структуру (векторами, матрицами, последовательностями, деревьями, графами).

Рассмотрим типы данных со структурными значениями. **Структурные значения** — это упорядоченные систематически организованные совокупности других (быть может, тоже структурных) значений, рассматриваемое как единое целое. Компоненты структурных значений также называется его **полями** или **элементами**.

Типичным примером структурного типа является комплексный тип, состоящих из двух вещественных полей: вещественной и мнимой частей.

Структурный тип характеризуется типом (или типами) своих компонент, и способом их организации (методом структурирования). Существует несколько методов структурирования, отличающихся способом доступа к компонентам и способом обозначения этих компонент: регулярный метод с индексированным или секвенциональным доступом к элементам, комбинированный метод с квалифицированным доступом к полям и др. **Регулярный** структурный тип содержит компоненты одного типа, называемого **базовым**. Напротив, **комбинированный** структурный тип содержит компоненты различных типов. Различают **индексированный** (компоненты структурного типа идентифицируются т. н. **индексом** (некоторым атомарным значением перечислимого типа прямо адресующим элемент), **квалифицированный** (компонента идентифицируется именем) и **секвенциальный** (последовательный) методы доступа.

Переменные и константы структурного типа называются **структурными**. Заметим что структурные константы отсутствуют в стандарте языка Паскаль, так что для этой цели используются структурные переменные, компоненты которых предварительно заполняются требуемыми значениями.

доступ\структура	регулярная	комбинированная
индексированный	массив	
секвенциональный	файл	
квалифицированный		запись

3. Составить ДТ проверки палиндромии слова

Билет 23(6)

1. Информация и сообщения. Интерпретация сообщения

2. Обобщенная инструкция композиции

3. Транспонирование квадратной матрицы 3x3 в ДТ

Билет 24 кто-нибудь знает?

Билет 25

1. Понятие файла(билет 3)

2. Эквивалентность программ и диаграмм(билет 7)

3. Умножение многочленов на Си

26 билет:

1)Автоматическая обработка информации

Чтобы выполнить автоматическую обработку информации, нужно располагать тремя физ. Представлениями

1)D – множество исходных данных

2) D' - Множество результирующих данных

3)Обработка из D в D' будет Р и оно будет выполняться на физ. Устройстве. Таким образом, для того, чтобы выполнить автоматическую обработку сообщений, необходимо автоматизировать выполнение трех отображений:

1) Отображения C, которое позволяет представить сообщение N в множестве данных D

2) Отображения P, которое переводит элемент D в D'

3) Отображения декодирования Q, которое позволяет интерпретировать результат в элемент N'

2) Тип запись

Это комбинированный структурный тип с квалифицированным методом доступа. Комбинированность означает, что поля записи имеют различные типы. Квалифицированный доступ негибок и не вычислим, т.к. требует явного указания. С помощью него можно делать удобные и понятные, в частности программисту, структуры.

3) обратный код 16 числа НАМ

```
*0->F*
*1->E*
*2->D*
*3->C*
*4->B*
*5->A*
*6->9*
*7->8*
*8->7*
*9->6*
*A->5*
*B->4*
*C->3*
*D->2*
*E->1*
*F->0*
*->.
->*
```

Билет 27:

1) вычислимые функции(а или б)

Определение 2.5.1. Функция $f: (A_s^*)^n \rightarrow A_t^*$ называется **вычислимой по Тьюрингу** (ВТ-функцией), если существует МТ с рабочим алфавитом A_p , содержащим алфавиты A_s и A_t (в силу принятого соглашения об алфавитах (см. п.2.6.1) для этого достаточно, чтобы $p = \max(s, t)$) такая, что функция $f_T : (A_s^*)^n \rightarrow A_t^*$, определяемая этой МТ, совпадает с f ($f_T \equiv f$). О любой такой МТ говорят, что она вычисляет f .

Из определения 2.4.3 в частности, следует, что если МТ T' моделирует машину T , то эти машины вычисляют одну и ту же функцию f .

Класс ВТ-функций определяет границы возможностей программирования: для невычислимых функций составление алгоритма невозможно!

2) Тип массив

Массив – регулярный структурный тип с индексированным методом доступа. Т.е. все элементы одного типа и все они имеют индекс. Для массивов одного типа определена операция присваивания ($A=B$) без поиндексированного копирования каждой переменной. Обработка массива в основном осуществляется покомпонентно, с помощью циклов.

3) НАМ сложение двух двоичных чисел

```
*+-#+*1c>c0
```

```
0c>1
c>1
1a>a1
0a>a0
+a>a+
1b>b1
0b>b0
+b>b+
0#a->#1
0#b->#0
1#b->#1
1#a->c#0
```

1->1
0->0

1*>a*
0*>b*
0#>#0
1#>#1
#>
+*>.
->*

28 билет:

1. Теорема о композиции (сто.82)

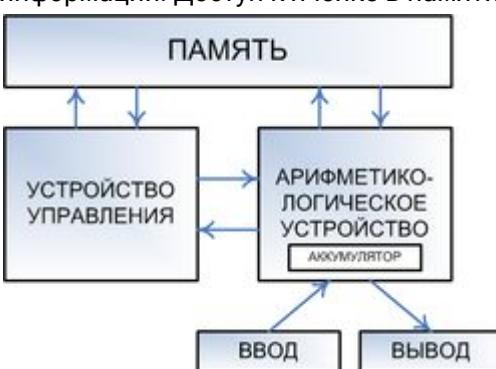
25) Теорема о композиции. Пусть n -местная функция $f : (A_s^{\nu}) \rightarrow A_t^{\nu}$ определяется равенством $f(u_1, u_2, \dots, u_n) = h(g_1(u_1, u_2, \dots, u_n), g_2(u_1, u_2, \dots, u_n), \dots, g_m(u_1, u_2, \dots, u_n))$, где g и h

ВТ-функции. Тогда f является ВТ-функцией, причем МТ, вычисляющую функцию f может быть эффективно построено из МТ, вычисляющих функцию g ($i=1\dots m$) и h

2. Модель фон Неймана. Имена и адреса. (стр 113)

Модель фон Неймана широко известный принцип совместного хранения команд и данных.

Память разделена на ячейки, в которых хранятся двоичные данные и команды, именуемые единицами информации. Доступ к ячейке в памяти осуществляется по адресу.



3. Си: лексикографическая упорядоченность последовательности слов

БЕЗ ПРОВЕРКИ СЛОВ, НАЧИНАЮЩИХСЯ НА ОДНУ И ТУ ЖЕ БУКВУ

```
#include <stdio.h>
#include <ctype.h>
#define OUT 0
#define IN 1
unsigned int char_to_set(char c) {
    c=tolower(c);
    if (c< 'a'||c>'z')
        return 0;
    else return 1u<<(c - 'a');
}
int main () {
    int alpha=1;
    int c, state, sim;
    state = OUT;
    unsigned int min = char_to_set('a');
    while ((c=getchar())!=EOF){
        sim=char_to_set(c);
        if (state==OUT&&sim!=0){
            if(sim>=min) min = sim;
            else {alpha=0;break;}
            state = IN;
        }
    }
}
```

```

if (state==IN&&(c=='\n'||c=='\t'||c==' '||c==',')) state=OUT;
}
if (alpha==1) printf("Упорядочена");
if (alpha==0) printf("НЕ упорядочена");
}

```

Билет 29

1. Теорема о ветвлении (см. билет 21.2)

2. Понятие типа данных

Тип данных – множество изображений(Сам объект), для которых определено правило их интерпретации, позволяющее каждому изображению сопоставить его значение, и множество атрибутов(Что с этим объектом делать), которые позволяют одному или нескольким элементам типа данных сопоставить либо изображения данных того же типа, либо изображения данных другого типа.

3. Инкремент 16-чного числа на НАМ

```

*A -> A*
* -> @
0@ -> 1!
1@ -> 2!
2@ -> 3!
3@ -> 4!
4@ -> 5!
5@ -> 6!
6@ -> 7!
7@ -> 8!
8@ -> 9!
9@ -> a!
a@ -> b!
b@ -> c!
c@ -> d!
d@ -> e!
e@ -> f!
f@ -> @0
@ -> 1!
! -> .
-> *

```

```

*0->0*
*1->1*
*2->2*
*3->3*
*4->4*
*5->5*
*6->6*
*7->7*
*8->8*
*9->9*
*A->A*
*B->B*
*C->C*
*D->D*
*E->E*
*F->F*
*->#
0#->.1
1#->.2
2#->.3
3#->.4
4#->.5
5#->.6
6#->.7

```

7#->.8
8#->.9
9#->.A
A#->.B
B#->.C
C#->.D
D#->.E
E#->.F
F#->#0
##->!1
!->.
->*

Рабочий вариант -

0->0
1->1
2->2
3->3
4->4
5->5
6->6
7->7
8->8
9->9
A->A
B->B
C->C
D->D
E->E
F->F
0*>.1
1*>.2
2*>.3
3*>.4
4*>.5
5*>.6
6*>.7
7*>.8
8*>.9
9*>.A
A*>.B
B*>.C
C*>.D
D*>.E
E*>.F
F*>*##
##->!1
!->.
->*

30 билет

1) теорема о цикле

27) Теорема о цикле. Пусть g – ВТ функция, вычисляемая функцией E , а p – ВТ предикат, вычисляемый машиной T . Тогда функция f , определяемая вычислительным процессом:

$$g = g_0, g_0 \in A_8^{\omega}; f(u_1, u_2, \dots, u_n) = g(u_1, u_2, \dots, u_{j-1}, g, u_{j+1}, \dots, u_n) \text{ пока}$$

$$p(u_1, u_2, \dots, g, \dots, u_n) = I, \text{ тоже является ВТ-функцией.}$$

28) Обобщенная теорема о цикле. Пусть $g_i: (A_8^n)^n \rightarrow A_8^{\omega} (i=1,2,\dots,m)$ – ВТ-функции,

вычисляемые МТ T_{gi} , а $p_i: (A_8^n)^n \rightarrow [I, L] (i=1,2,\dots,m)$ – ВТ предикаты,

вычисляемые МТ $T_{pi} (i=1,2,\dots,m)$ соответственно, тогда функция, определяемая соотношениями:

$$g_i: (A_8^n)^n \rightarrow A_8^{\omega} (i=1,2,\dots,m),$$

$$f(u_1, u_2, \dots, u_n) = \begin{cases} g_1(u_1, u_2, \dots, u_{j-1}, g, u_{j+1}, \dots, u_n), & \text{если } p_1(u_1, \dots, g_1, \dots, u_n) = I \\ g_2(u_1, u_2, \dots, u_{j-1}, g, u_{j+1}, \dots, u_n), & \text{если } p_2(u_1, \dots, g_2, \dots, u_n) = I \\ \dots \\ g_m(u_1, u_2, \dots, u_{j-1}, g, u_{j+1}, \dots, u_n), & \text{если } p_m(u_1, \dots, g_m, \dots, u_n) = I \end{cases} \text{ пока}$$

$$\wedge j=1 \wedge m p_j = I$$

Также является ВТ-функцией, причем МТ, вычисляющая функцию, может быть эффективно построена из машин $T_{gi} \cup T_{pi} (i=1,2,\dots,m)$

2) процедуры и функции (в методичке стр 185)

3) написать на си конвертацию тьюринговской машины из пятерок в четверки

Компилировать через gcc, на предупреждения внимание не обращать.

```
#include <stdio.h>
```

```
int main() {
    int q0, q1;
    char a, b, move;
    while (scanf("%d,%c,%c,%c,%d", &q0, &a, &b, &move, &q1)) {
        if ((int)move == (int)'u') {
            printf("(%d,%c,%c,%d)\n", q0, a, b, q1);
        }
        else if ((int)a == (int)b) {
            printf("(%d,%c,%c,%d)\n", q0, a, move, q1);
        }
        if ((int)move == (int)'s') {
```

```

        break;
    }
}
else {
    printf("(%d,%c,%c,%d)\n", q0, a, b, q0 + 100);
    printf("(%d,%c,%c,%d)\n", q0 + 100, b, move, q1);
}

```

31 билет

1. Схемы Тьюринга. Нисходящая разработка

Нисходящая разработка состоит в следующем. Мы функцию f выражаем через f_1, f_2, \dots, f_k и составляем диаграмму машины Т, включающую символы машины T_i ($i=1\dots k$). Каждую из функций f_i выражаем через новые, более простые. И так до тех пор, пока на каком-то уровне не получатся диаграммы, включающие только символы элементарных МТ

2. Критика языков Паскаль и Си

1) Язык Паскаль представляет собой программно-компилируемую реализацию машины фон Неймана и к нему можно отнести всю критику этой алгоритмической модели. Скалярный оператор присваивания, через бутылочное горлышко которого надо прокачивать сложные математические объекты, заставляет программиста постоянно заботиться о рациональном использовании этой шины ($:=$) и отвлекает от решения самой задачи. Справедливости ради отметим, что эта порочная система в прошлом, настоящем и ближайшем будущем остается единственным технически возможным средством автоматизации обработки информации. Язык Паскаль является строго типизированным языком. Все объекты программ на Паскале должны быть описаны и употребляются в строгом соответствии с описаниями. Это предполагает строгую дисциплину программирования, неудобную в задачах системного программирования. Однако в своей строгости Паскаль непоследователен. Например, записи с вариантными частями образуют брешь в системе типизации, не уступающую тому самому метру государственной границы, о котором мечтал герой сатирического романа Остап Бендер. Тем не менее, особенности Паскаля таковы, что возможна реализация быстрого однопроходного эффективного компилятора. Языковая среда Паскаля также невелика, проста и эффективна. Паскаль содержит только такие языковые средства, которые эффективно компилируются на аппаратуру. Но это превращается в недостаток, потому что многие нужные любому программисту типы данных (такие как строки и массивы переменной длины) либо не реализованы вообще, либо их (эффективная) реализация наталкивается на ряд проблем. Модель идеального компьютера, представляемая Паскалем, слишком далека от современной программно-аппаратной среды, представляемой современными ОС. Товарное программирование на Паскале невозможно ввиду отсутствия модульности и внешних процедур. Паскаль проигрывает Си не только в выразительной силе и лаконичности. Библиотека языка Си представляет собой весьма мощную интерпретируемую компоненту.

2) Не только критики, но и приверженцы Си признают, что этот язык весьма сложен и наполнен опасными элементами, которые очень легко использовать неправильно. Своей структурой и правилами он никак не поддерживает программирование, нацеленное на создание надёжного и удобного в сопровождении программного кода, напротив, рождённый в эпоху прямого программирования под различные процессоры, язык способствует написанию небезопасного и запутанного кода. Многие профессиональные программисты склонны считать, что язык Си – мощный инструмент для создания элегантных программ, но в том же время с его помощью можно создавать крайне некачественные решения.

Из-за различных допущений в языке программы могут компилироваться с множественными ошибками, что часто приводит к непредсказуемому поведению программы. Современные компиляторы предоставляют опции для статического анализа кода, но даже они не способны выявить все возможные ошибки. Результатом неграмотного программирования на Си могут стать уязвимости программного обеспечения, что может оказаться на безопасности его использования.

У Си достаточно высокий порог вхождения, что затрудняет его использование в обучении в качестве первого языка программирования. Наконец, за более чем 40 лет существования, язык успел несколько устареть, и в нём достаточно проблематично использовать многие современные приёмы и парадигмы программирования.

Если по пунктам, то:

1. Примитивная поддержка модульности
2. Предупреждения вместо ошибок
3. Высокий порог вхождения
4. Отсутствие контроля инициализации переменных
5. Отсутствие контроля над адресной арифметикой
6. Динамически выделяемая память
7. Неудобные и небезопасные нуль-терминированные строки
8. Небезопасная реализация функций с переменным числом аргументов
9. Отсутствие унификации обработки ошибок
3. Написать программу, выводящую график функции на Си

32 билет

1) Обобщенная теорема о цикле.

Просто теорема о цикле

Применение циклов при конструировании машин Тьюринга обосновывается следующей теоремой.

Теорема 2.6.4. Пусть $g : (A_s^*)^n \rightarrow A_s^*$ – ВТ-функция, вычисляемая машиной T_g , а $p : (A_s^*)^n \rightarrow \{\text{И}, \text{Л}\}$ – ВТ-предикат, вычисляемый машиной T_p . Тогда функция $f : (A_s^*)^n \rightarrow A_s^*$, определяемая вычислительным процессом

$$g = g_0, g_0 \in A_s^*$$

$$f(u_1, u_2, \dots, u_n) = g(u_1, u_2, \dots, u_{j-1}, g, u_{j+1}, \dots, u_n),$$

пока $p(u_1, u_2, \dots, g, \dots, u_n) = \text{И}$,

тоже является ВТ-функцией, причем МТ, вычисляющая функцию f , может быть эффективно построена из машин T_g и T_p .

Теорема 2.6.6. (об обобщённом цикле, Э. Дейкстра) Пусть $g_i : (A_s^*)^n \rightarrow A_s^*$ ($i = 1, 2, \dots, m$) — ВТ-функции, вычисляемые МТ T_{g_i} , а $p_i : (A_s^*)^n \rightarrow \{\text{И}, \text{Л}\}$ ($i = 1, 2, \dots, m$) — ВТ-предикаты, вычисляемые МТ T_{p_i} ($i = 1, 2, \dots, m$) соответственно. Тогда функция, определяемая соотношениями:

$$g_i = g_{0_i}, g_{0_i} \in A_s^*, i = 1, 2, \dots, m$$

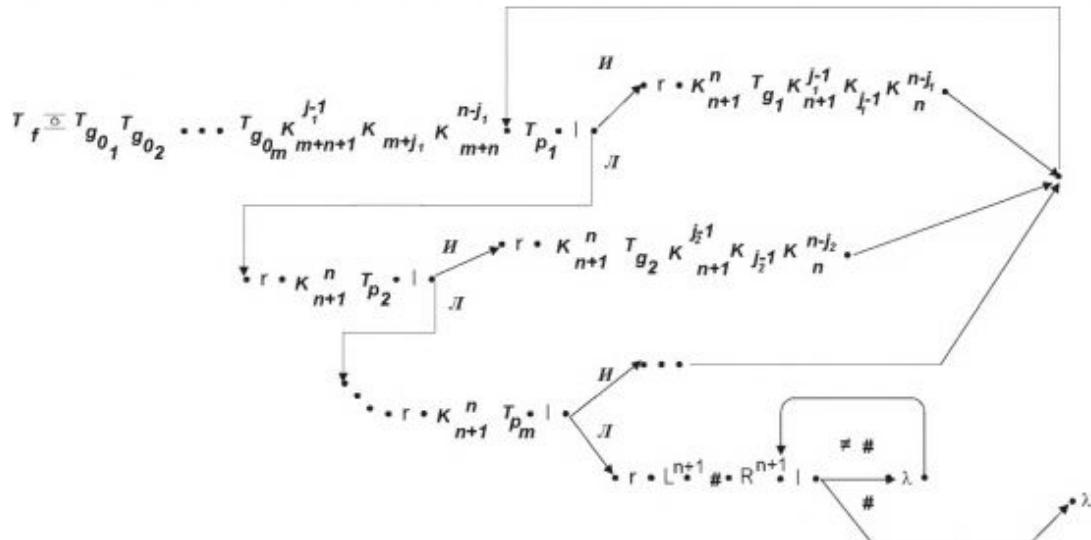
$$f(u_1, \dots, u_n) =$$

$$\begin{cases} g_1(u_1, \dots, u_{j_1-1}, g_1, u_{j_1+1}, \dots, u_n), \text{ если } p_1(u_1, \dots, g_1, \dots, u_n) = \text{И}, \\ g_2(u_1, \dots, u_{j_2-1}, g_2, u_{j_2+1}, \dots, u_n), \text{ если } p_2(u_1, \dots, g_2, \dots, u_n) = \text{И}, \\ \dots \\ g_m(u_1, \dots, u_{j_m-1}, g_m, u_{j_m+1}, \dots, u_n), \text{ если } p_m(u_1, \dots, g_m, \dots, u_n) = \text{И} \end{cases}$$

$$\text{пока } \bigvee_{j=1}^m p_j = \text{И},$$

тоже является ВТ-функцией, причем МТ, вычисляющая функцию, может быть эффективно построена из машин T_{g_i} и T_{p_i} ($i = 1, 2, \dots, m$).

▷ Машина T_f , вычисляющая функцию f , определяется диаграммой:



Машины $T_{g_{0_i}}$ ($i = 1, 2, \dots, m$) аналогичны машине T_{g_0} , рассмотренной при доказательстве теоремы 4.5.5. Теорема доказана. \square

Замечание. Данная теорема описывает цикл с m альтернативными телами. Также как и в случае ветвления, каждое тело охраняется своим предикатом. Но в отличие от ветвления, цикл ориентирован на многократное повторение, причем в данном случае повторения могут идти по разным телам. Поэтому требования к набору предикатов снижаются. Отказ всех предикатов цикла в отличие от ветвления не приводит к аварийному завершению, а всего лишь прекращает выполнение цикла. («Что русскому хорошо, то немцу смерть».) Если набор предикатов цикла противоречив и при каком-либо повторении цикла соответствующими предикатами открыто для выполнения более одного тела, то выполняется любое из этих тел. То есть в данном случае программируется недетерминированное выполнение, выводящее нас за пределы алгоритма в традиционном понимании. В простейшем случае прямого исполнения такого цикла на последовательном компьютере может быть выполнено первое в порядке написания тело. Таким образом цикл никогда не отказывает. При реализации цикла необходимо учитывать его двойственную природу: стремление продолжать выполнение цикла с запрограммированным завершением, т. е. предикаты p_i должны зависеть от результатов вычисления функций g_k на предыдущих итерациях так, чтобы они постепенно закрывали тела цикла, приводя к его завершению. Ведь цикл работает «до последнего предиката»!

Модель фон Неймана пришла на замену модели МТ, дабы избавиться от недостатков предыдущей модели. К модели фон Неймана относят как аппаратные компьютеры, так и традиционные языки программирования. Их математические основания считаются сложными, громоздкими и концептуально бесполезными. Модель фон Неймана использует память и чувствительна к предыстории. Семантика также заключается в переходах из состояния в состояние, только они более сложные. Ясность программ гораздо выше. Наибольшая проблема модели фон Неймана – шина, которая за один раз может передавать только один определенный элемент памяти и является «Узким горлышком». Таким же горлышком в программировании является оператор присваивания, именно он вынуждает нас программировать на уровне «слово за слово».

3) Написать на Си табуляцию многочлена и его производной на отрезке.

```

double diho (double a, double b, func F){
    double as=0;
    double bs=0;
    while ((fabs(a-b)>eps)) {
        if ((F(a)*F((a+b)/2))>0)
        {
            as=(a+b)/2;
            bs=b;
        }
        if ((F(b)*F((a+b)/2))>0)
        {
            as=a;
            bs=(a+b)/2;
        }
        a=as;
        b=bs;

    }
    return ((a+b)/2);
}

double iteration (func XF, func d_F, double a, double b, double eps){
    double zeroX = (a+b)/2, x = a, K=0;
    if (iteration_check(d_F,a,b)==1) return 0;
    while (fabs(x-zeroX)>eps) {
        K=zeroX;
        zeroX=x;
        x=XF(K);
    }
    return x;
}

double NewTone (double F(double), double d_F(double), double a, double b, double eps){
    double x=(a+b)/2;
    double x1=x+1;
    while(fabs(x-x1)>eps) {

```

```
x1=x;
x=x-F(x)/d_F(x);
}
return x;
}
```