КАЗАНСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Кафедра технологий программирования

Н.Р. БУХАРАЕВ

ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ

Учебно-методическое пособие для подготовки к вступительному экзамену по информатике для поступающих в магистратуру ИВМИТ КФУ.

Казань – 2018

УДК: 004.42 Программирование на ЭВМ. Компьютерные программы

ББК: 32.81.73

Принято на заседании кафедры технологий программирования
Протокол № 3 от 15 ноября 2018 года

Рецензенты:
кандидат физико-математических наук,
доцент кафедры технологий программирования КФУ **А.И.Еникеев;**
кандидат технических наук,
доцент кафедры технологий программирования КФУ **А.М.Гусенков**

Бухараев Н.Р.
Введение в процедурное программирование. /Н.Р.Бухараев. – Казань: Казан. ун-т, 2018. – 99 с.

Предлагаемое пособие адресовано поступающим в магистратуру ИВМИТ КФУ и соответствует списку вопросов вступительного экзамена по информатике, по состоянию на 2018 г.

© Бухараев Н.Р., 2018
© Казанский университет, 2018

СОДЕРЖАНИЕ

От составителя. 4

Вопросы экзамена 6

§1. Основные понятия процедурного программирования. 7

§2. Введение в технологию программирования. 17

§ 3. Классификация типов данных. 24

§4. Вычисление предикатов. 30

§5. Поиск. 35

§ 6. Упорядоченные типы. 41

§ 7. Сортировка. 45

§8. Машинно-ориентированное программирование. 48

§ 9. Абстрактные линейные типы. 55

§10. Алгоритмы полного перебора. 67

§ 11. Перебор с возвратом. 71

§12. Нелинейные типы данных. 73

§ 13. Алгоритмы обработки выражений. 80

§ 14. Алгоритмы текстовой обработки. 91

§ 15. Формальные вычисления. 96

Литература 99

От составителя.

Предлагаемое пособие соответствует списку вопросов вступительного экзамена в магистратуру ИВМИТ по информатике (по состоянию на 2018 г.).

Однако *не* стоит видеть в нем конспект готовых ответов (или, попросту - шпаргалок). Удачные примеры такого рода автору неизвестны. И потому данное пособие задумывалось как *связанный* и *замкнутый* по форме текст, призванный дать пищу *собственным* размышлениям и придать импульс к *собственной* работе.

Посвященный каждому вопросу программы параграф использует понятия, введенные в предыдущих ответах, а решение задач - ссылки на решение задач из предыдущих ответов. Теоретически можно и в идеале нужно (фактически же – лишь по запросу экзаменатора) уметь "разворачивать" ответ до базовых общематематических понятий из первого параграфа. Потому, не удивляйтесь - при приблизительно равном фактическом объеме ответов, более поздние разделы выглядят компактнее более ранних.

Что оценивается на экзамене? *Не* ваша программистская интуиция, и *не* знание конкретных языков или систем программирования - но ваша квалификация как программистов, уже окончивших *университетский* курс обучения. По факту, это означает умение уверенно создавать пусть не идеально эффективные, но всегда *«грамотные, понятные* и потому *правильные* и *надежные* программы. Последнее предполагает свободное владение методологией решения достаточно сложных задач - невозможного без *фундаментальной* общематематической подготовки, умения формулировать *вне* синтаксиса конкретного языка программирования *точную* семантику используемых понятий в терминах классических понятий.

При том нужно ясно осознавать, что задача экзаменатора - вовсе не в том, чтобы "поймать" вас на отдельных формулировках, но в том, чтобы оценить вашу программистскую и общематематическую культуру *в целом*. Потому - не старайтесь заучивать приведенные формулировки наизусть - поздно и

бесполезно! Лучше задумайтесь - как бы *вы сами* точно и полезно сформулировали содержание используемых в вашей программистской практике понятий. Надеюсь - ваши формулировки будут близки к моим. Успеха!

Выделения в тексте имеют следующий смысл. Понятия, выделяемые ***жирным шрифтом*** необходимо раскрыть в ответе, *курсивом* - желательно обратить внимание, мелким шрифтом - можно опустить, без ущерба для итогового результата.

Вопросы экзамена

1. Основные понятия процедурного программирования.

2. Пользовательские процедуры как аппарат технологии программирования.

3. Типы данных и их классификация (на примере языка Паскаль).

4. Алгоритмы вычисления логических формул.

5. Алгоритмы поиска в последовательностях.

6. Однопроходные алгоритмы объединения (слияния), пересечения и разности массивов.

7. Простые алгоритмы сортировки массивов.

8. Реализация операторов и типов данных средствами низкого уровня.

9. Списки, стеки, очереди и их применение.

10. Алгоритм полного перебора на примере задачи о перечислении всех правильных раскрасок графа.

11. Алгоритм перебора с возвратом на примере задачи о перечислении всех правильных раскрасок графа.

12. Обход дерева "в глубину" (с использованием стека) и "в ширину" (с использованием очереди).

13. Алгоритмы обработки арифметических выражений.

14. Определение и реализация основных операций обработки текстов.

15. Нахождение текста результата операции по тексту ее аргументов на примере двух способов вычисления суммы натуральных чисел.

Более детально рамки вопросов и ответов определяют явно выделяемые далее базовые понятия и указания на демонстрируемые умения и навыки.

§1. Основные понятия процедурного программирования.

Ключевые понятия: состояние, оператор, типы данных. Структурное программирование – композиция, условные и итерационные (циклические) определения операторов. Рекуррентные определения.

Умения и навыки: Преобразование неструктурных алгоритмов в структурные. Нахождение рекуррентных определений кратных числовых функций. Преобразование рекуррент в итерацию.

При моделировании реальных объектов мы выделяем описание *статики* (одновременности) – допустимого положения, характеристики, состояния объекта на некоторый фиксированный момент времени - и *динамики* - изменения объекта во времени, допустимых преобразований его состояний. Формально-математическим средством такого описания служит понятие *области* (модели, домена, системы, пространства) или, в программистской терминологии, (абстрактного) ***типа данных*** - пары <A,F>. Здесь A - некоторое основное множество значений, отражающее совокупность допустимых состояний объекта, а F, $F \subseteq A \to A$, - класс функций, описывающих совокупность его допустимых преобразований во времени. (Здесь и далее $X \to Y$ - класс функций f с областью определения Dom(f)=X и областью значений Val(f)=Y, не обязательно - всюду определенных). Сам термин «тип» указывает на то, что на деле такое описание не задает конкретный, содержательно богатый объект однозначно, но описывает некоторое зависящее от цели моделирования обеднение, математическую *абстракцию объекта.*

Помимо формулирования цели программирования как создания компьютерных моделей, понятие типа используется и для описания языка программирования как основного средства такого моделирования. Здесь для

программиста явное выделение класса функций F особенно важно, поскольку оно задает базис - класс предопределенных, *доступных* для использования преобразований. В отличие от обычной математической практики, класс таких исходных функций, равно как и способы определения новых функций в программировании крайне ограничены.

Понятие типа помогает нам точнее сформулировать и задачу самого программирования как перевод, описание допустимого в терминах доступного.

В наиболее простом случае элементы множества A считаются "атомами", не имеющими, в контексте данного языка описания, внутренней структуры - возможных/допустимых значений некоторой единственной характеристики объекта, отождествляемой с ним самим. Такие типы называют простыми или ***скалярными.***

Классические примеры типов, традиционно воспринимаемых нами как скалярные - арифметика целых Z, рациональных Q и вещественных D чисел, т.е. соответствующие числовые множества, рассматриваемые совместно с 4 арифметическими операциями и отношениями сравнения. Универсальность математических понятий позволяет использовать одни и те же пространства для описания содержательно различных характеристик объекта - так, вещественные числа могут пониматься как значения скорости, ускорения, веса, температуры и прочих характеристик объекта.

В отличие от классического разделения «скаляры-векторы», в программировании понятие простого и сложного относительны. Тип, считающийся скалярным в одном языке программирования, может трактоваться как структурный – в другом.

В более сложных случаях описания статики объекта требуется несколько характеристик - каждая, вообще говоря, со своим собственным множеством значений. В классической математике для описания состояния сложного объекта как совокупности значений его характеристик обычно используется понятие векторного пространства.

Т.е. некоторого множества n-ок $p=<p_1,..,p_n>$, $p_i \in A_i$, p_i - значение i-ой характеристики (как правило, числовое). В этом случае основное множество типа – n-местное отношение, подмножество *декартового произведения* $A_1 \times \ldots \times A_n$ (множества всех n-ок $p=<p_1,..,p_n>$, $p_i \in A_i$, $i \in [1..n]$). Так, множество точек плоскости можно описать как декартово произведение $D^2=D\times D=\{<x,y>, x\in D, y\in D\}$, где x,y - декартовы координаты точки, любую фигуру на плоскости - как некоторое подмножество D^2. Другой способ описания того же пространства - задание точек парой значений их полярных координат.

В программировании вместо позиционного принципа, требующего предварительной, часто неявной, нумерации характеристик, обычно используется явное именование - сопоставления значений не номерам, но *именам* характеристик. В этом случае в качестве основного множества типа выступает подмножество ***именованного декартового произведения*** *множеств* $A_1,\ldots,A_n$ - класса всех *функций значения*, т.е. определенных на некотором множестве имен Name всех функций f таких, что $f(k)\in A_{i(k)}$, $k \in$ Name.

Пример. Множество точек плоскости можно описать как именованное декартово произведение - множество всех функций f на множестве имен {'X-координата', 'Y-координата'}, где f('X-координата') ∈ D, f('Y-координата') ∈ D. При задании точек полярными координатами мы, по всей видимости, выберем уже другое множество имен - например, {'радиус', 'угол'}, что помогает явно различить разные представления понятий.

Отметим, что хотя в качестве имен обычно используются слова (последовательности символов), наше определение позволяет понимать под именами *любое* множество значений - например, снова - целые числа (индексы). В случае использования таких "нетрадиционных" имен обычно предпочитают говорить не об именах, а о *ссылках* на значение. На практике различие между именованным и "просто" декартовым произведениями довольно условно. Как правило, мы стараемся описать объект как упорядочиваемый (по меньшей мере, одним способом) набором "координат", т.е. независимых линейных характеристик (см. "Классификация типов").

Возможность описания противоположных по содержательному смыслу понятий - статика и динамика, значения и преобразования - в рамках одного языка функций - одно из наиболее фундаментальных положений программирования.

Итак, в общем случае типа данных <A,F> в качестве множества состояний A выступает некоторое множество функций, а F - некоторый класс функций над A, $F \subseteq A \rightarrow A$. Функции "высших порядков", аргументами и значениями которых являются функции, называют *операторами,* а их определение на *некотором* формально-математическом языке называют - ***спецификацией.*** Классический способ спецификации – задание областей определения («дано») и значений («требуется найти/вычислить») оператора парой предикатов, называемых ее *пред- и пост-условием.*

Примеры см. далее. На практике мы часто подразумеваем тривиальные предусловия, не формулируя их явно.

Как следует из названия, процедурное программирование изначально ориентировалось на функциональное описание динамики - сложных преобразований, имеющих «вход» и «выход», начало и конец, аргументы и результаты – в операторных терминах.

Язык процедурного программирования - язык формального описания реальных процессов и объектов в терминах задаваемых этим языком способов определения типов данных и процедур их преобразования.

Языки ***процедурного программирования*** - *языки прямых определений,* задающие способы определения (порождения, конструирования) новых операторов в терминах (уже построенных к этому времени) типов данных. Эти определения операторов мы и будем называть процедурами.

Это существенно отличает их от языков аксиоматической математики и рекурсивного (функционального и логического) программирования, оперирующих *косвенными* определениями понятий указанием их свойств, в частности – в терминах рекурсивных определений.

Базовый *рекуррентный* принцип ("новое" определяем через "старое") отражается в основном для таких языков общем способе обозначения операторов – ***операторе*** кратного или векторного ***присваивания вида s:=e(s').*** Здесь s, s' - списки имен переменных, а e - выражение, задающее функциональную зависимость новых значений переменных (с именами из s) от старых значений переменных (с именами из s'). Существенно, что имена в s, s' могут *совпадать*.

Важные частные случаи присваивания, используемые (*в отличие* от кратного присваивания) в реальных языках программирования:

a) s:=s', где s, s' - произвольные списки имен переменных (такое присваивание соответствует *табличному определению* некоторой функции, в частном случае - *перестановке значений*)
b) *простое* присваивание вида v:=e(s), v - (единственное) имя переменной
c) *бинарное* присваивание вида v:=[vf]v', v,v' - имена переменных, f - имя (знак) функции двух аргументов.

В контексте процедурного программирования, ***программа*** трактуется как процедура, итоговое определение оператора на *заданном* языке программирования по спецификации – определению этого оператора на некотором формальном языке, а ***программирование*** - такое определение как происходящий во времени *процесс*.

Как отмечено выше, с абстрактной, общематематической точки зрения и значения (из A), и их преобразования (из F) - это *одни и те же* объекты (функции), *с одними и теми же* способами их определения. С другой, для программиста определение типов как пары <данные, преобразования> также отражает фундаментальное разделение *хранимой* и *выводимой* информации.

В программировании значения переменных трактуются как содержимое ячеек памяти, с существенным разделением на память внутреннюю и внешнюю по отношению к устройству вычисления. Это - быстрая, но малоемкая *оперативная* память и файлы – более медленная, но емкая память, связанные с внешними (периферийными) устройствами ввода-вывода. Более точно, программа - определение *файловой* процедуры (или как процедуру обработки *внешних потоков данных* - что еще точнее, но требует явного уточнения интерактивного подхода).

Классическим примером конструктивного построения языка являются языки ***структурного программирования,*** выделяющие в качестве основных три общематематических способа (структуры, схемы) определения новых процедур:

а) ***композиция*** – (S1;S2, в синтаксисе языка Паскаль), соответствующая описанию последовательного вычисления процедур S1 и S2 как функций на состояниях: (S1;S2)(s)=S2(S1(s))

Композиция операторов соответствует определению «сложных функций» подстановкой (суперпозицией).

b) ***структуру условного оператора*** (в Паскале - if B then S1 else S2), соответствующую определению функции разбором случаев

$$(\text{if B then S1 else S2})(s)=\begin{cases} S1(s), & B(s)=\text{true} \\ S2(s), & B(s)=\text{false} \end{cases}$$

В классических терминах, соответствует условному определению функций.

c) ***структуру оператора цикла с предусловием*** (в Паскале - while B do S), соответствующую итогу вычисления первого члена рекуррентно определяемой последовательности s, не обладающим заданным свойством B. Точнее:
   a) если $s_0=s$, $s_{i+1}=S(s_i)$ и $k=\min\{i: B(s_i)=\text{false}\}$, то (while B do S)(s)=$s_k$
   b) если $\{i: B(s_i)=\text{false}\}=\varnothing$, то значение (while B do S)(s) не определено,

В классических терминах означает итерационное определение функции.

Семантика *всевозможных* структур управления задается языком (произвольных) блок-схем или, эквивалентно, понятием ссылки на оператор (оператор goto). Важный факт заключается в том, что ограниченных средств структурного программирования достаточно для определения всех процедур, с точностью до ***функциональной эквивалентности***, т.е. равенства классов определяемых функций.

Пример структурирования программ

Задача линейного поиска значения в последовательном файле. Неформальная постановка – выяснить (found), есть ли заданное значение x в заданной последовательности a. Спецификация: предусловие $A\in N\rightarrow T$ and $X\in T$, постусловие found=$\exists\ i\in \text{Dom}(A)(X=A_i)$:

Здесь и далее мы обозначаем значения большими буквами, чтобы явно отличить их от имен значений.

// не структурный, не удачный вариант решения

read(x); found:=false; while not eof() do begin read(a); if a=x then begin found:=true; goto 1 end; 1: ≈

// структурный, лучший вариант

read(x); found:=false; while not eof() and not found do begin read(a); if a=x then begin found:=true; end;

Если понятие функциональной эквивалентности процедур описывает начало и конец вычислений (вход и выход, т.е. состояния - аргументы и результаты процедур), то кратное применение принципа прямых определений приводит к понятию ***трассы*** (следа) как последовательности промежуточных состояний вычисления, применяемое для описания процесса вычислений. Определение множества трасс - ***трассировка -*** играет существенную роль для понимания исполнения и доказательства корректности процедур.

Нетрудно увидеть за этими определениями хорошо знакомое различие между двумя определениями последовательности – реккурентным (зависимость от *предыдущего* члена) и формулой общего члена, задающей зависимость члена последовательности от номера (и *первого* члена).

Пример трассировки для доказательства корректности программ. Задача обмена значений (вещественных) переменных, спецификация x,y:=y,x.

Программа	Трасса		Программа	Трасса			Программа	Трасса	
	x	y		x	y	z		x	y
	X	Y		X	Y	!		X	Y
x:=y;	Y	Y	z:=x;	X	Y	X	x:=x+y;	X+Y	Y
y:=x	Y	Y	x:=y;	Y	Y	X	y:=x-y;	X+Y	X
			y:=z	Y	X	X	x:=x-y	Y	X
неверно			верно				верно		

Примеры обратного перехода от непосредственного определения последовательностей (как функций на N, т.е. "формулы общего члена") к итерационным определениям.

Спецификация	Итерационное определение	Программа
$m:=\max(a_1,\ldots,a_n)$	$m_1= a_1$	read(m);
	$m_{i+1}= \max(m_i,a_i)= \begin{cases} a_i, & m_i<a_i \\ m_i, & m_i \geq a_i \end{cases}$	while not eof() do if m<a then m:=a
$s:=\Sigma\ \{a_i: i\in [1..k]\}$	$s_0=0$	s:=0; read(eps);
где	$s_{i+1}= s_i+a_i$	read(a); go:= true;

$k=\min\ \{i: abs(a_i)<eps\ \}$		while not eof() and go do
		begin read(a);
		if abs(a)>=eps
		then s:=s+a
		else go:=false
		end

Примечание. Общую теорию приближения числовых функций суммами (теорию рядов) - ищи в курсе мат. анализа.

$b:=a_0x^n+\ldots+a_nx^0$	$b_0= a_0$	read(b);
	$b_{i+1}=b_ix+a_i$	while not eof() do
	(схема Горнера)	begin read(a); b:=b*x+a end

Примечание. Замечательный - и достаточно редкий - случай наилучшего решения проблемы заданными средствами (здесь - с наименьшим количеством вычисления арифметических операций)

§2. Введение в технологию программирования.

Ключевые понятия. Параметры-переменные и значения. Глобальные и локальные объекты. Семантика обращений. Побочные эффекты.

Знания и умения. Демонстрация технологии нисходящего программирования на примере.

Возможны 2 естественных порядка (стратегии, принципа) определения процедур. Принцип композиции - от данных средств реализации (т.е. языка программирования ЯП) к желаемой цели (логике, языку спецификаций) - ***восходящее программирование*** – первоначально кажется наиболее естественным, поскольку следует порядку порождения процедур в ЯП и их последующего применения к данным (порядку вычисления). Однако, в силу связанного с таким порождением быстрого роста числа рассматриваемых случаев, такой подход, скорее - локальная тактика, применимая для разработки относительно небольших и несложных программ. Стратегия ***нисходящего программирования*** - цель (логика) важнее средств (реализации) - основана на принципе *декомпозиции*, последовательного перехода от определения на некотором исходном языке спецификации к более детальным определениям той же процедуры на все более бедных языках спецификаций. Это - чисто семантический принцип, не зависящий по существу от конкретного языка программирования. (В реальности же, как всегда, успех обеспечивает лишь сочетание стратегии и тактики!)

Термин "технология" предполагает наличие некоторого языка фиксации, описания этапов процесса собственного мышления при разработке сложной программы. Аппарат определения ***пользовательских типов данных и пользовательских процедур*** позволяет определять нужные языки спецификации средствами *самого* языка программирования, реализовывая

принцип *упреждающего именования* - возможности обращения, т.е. *формального* использования имен ("заголовков") функций (операторов) прежде, чем они фактически определяются как выражения языка программирования.

Определение пользовательской процедуры есть сопоставление пользовательскому имени функции (оператора) выражения ЯП, называемого *телом процедуры.* На уровне семантики, это разграничение определяют как различие между *интерфейсом* и *реализацией* процедуры.

Единственной - но критичной! - сложностью такого предварительного определения (объявления) имени является выделение (именование и определение типа) входных и выходных значений - (возможно, *совпадающих*) имен аргументов и результатов процедуры как функции а также, возможно**,** имен ***глобальных*** значений - т.е. его *параметров* как функции (в общематематическом, а не узко-программистском смысле термина). В реальности, существующая традиция предполагает выделение *непересекающихся* множеств имен - т.н. формальных ***параметров-значений*** - "собственно" входов, не являющихся выходами, и ***параметров-переменных*** - всех выходов, т.е. изменяемых значений. Имена глобальных объектов при этом никак не выделяются. Последующее фактическое определение тела процедуры, как правило, также требует введения вспомогательных объектов, удовлетворяющих ***локальным,*** т.е. временным определениям.

Пример порядка разработки при нисходящей стратегии (восходящая предполагает обратный порядок!).

Задача. Вычисление значения комплексного многочлена a степени Deg(a). Спецификация-постусловие (предполагает знание типа "арифметика комплексных чисел"): $y(a,x):=\Sigma\{a_i*x^i:i\in[1..Deg(a)]\}$, $x\in tComplex$, $\forall i\in[1,Deg(a)](a_i\in tComplex)$.

1) Решаем задачу в терминах ее постановки (спецификации). Объявляем (не определяя) имена нужных пользовательских типы и процедур.

```
type
tIndex=? ;              {номера коэффициентов}
tComplex=?;                  {комплексные числа}
tComplexPolynom=?      {комплексные многочлены}
procedure SetZero(var c:tComplex); {c:=0} ?
function Degree(p:tPolynom):tIndex; {определить степень полинома}?
function Element(p:tPolynom,i:tIndex):tComplex;{определить значение коэффициента}?
procedure Mult(a,b:tComplex;var c:tComplex); {c:=a  b} ?


procedure PolynomValue(Polynom: tComplexPolynom; var y: tComplex);
var
Coefficient:tComplex;    { коэффициент полинома }
i,                        {его номер}
DegA:tIndex;                {степень полинома}

begin
    SetZero(y);{y:=0}
    DegA:=Degree(Polynom);
    for  i:=1 to DegA do
    begin
        Coefficient:= Element(Polynom,i); { Coefficient := ai }
        Mult(y, Coefficient,y);{y:=y*x+ai;}
    end; {for}
end; {procedure PolynomValue }
```

Выделены операторы типа "арифметика комплексных чисел" - или, в более программистской терминологии, "подзадачи"

2) Даем фактическое определение пользовательских объектов - желательно такое, чтобы обеспечить наиболее эффективное вычисление основного алгоритма (при восходящем программировании, наоборот, реализация типов управляет их логикой). Если такое определение тривиально, можно, исходя из соображений эффективности, сразу подставить значение (а именно - модифицированное тело, см. далее) в исходный текст.

tIndex=1..100; {строго говоря - ограничиваем постановку}

tComplex=record x,y:real end;

tComplexPolynom=record Coefficient:array[tIndex] of tComplex; Degree:tIndex end;

procedure SetZero(c:tComplex); begin **c.x:=0;c.y=0** end;

{function Degree(p:tPolynom):tIndex; = **p.Degree** }

{function Element(p:tPolynom,i:tIndex):tComplex**;=p.Coefficient[i]** }

procedure Mult(a,b:tComplex;c:tComplex);

begin

c.x:=a.x*b.x-a.y*b.y;

c.y:= a.x*b.y+a.y*b.x;

end;

procedure PolynomValue(Polynom: tComplexPolynom; var y: tComplex);

var

{Coefficient:tComplex; = **Polynom.Coefficient[i]** }

i:tIndex; {номер коэффициента}

{DegA:tIndex; = **Polynom.Degree** }

begin

SetZero(y);{y:=0}

for i:=1 to **Polynom.Degree** do Mult(y, **Polynom.Coefficient[i]**,y);{y:=y*x+a_i;}

end; {PolynomValue }

Выделена реализация операторов языка спецификации. Понятно, реализовали полином "псевдодинамическим" массивом (а не, скажем файлом или списком) для того, чтобы быстрее и компактнее вычислить функции Degree

и Element. Несмотря на то, что пришлось фактически ограничить постановку исходной задачи.

Формальная семантика обращения к (не рекурсивным) процедурам может быть точно описана в терминах 3-х правил преобразования (изменения) тела процедуры - построения ***модифицированного тела процедуры -*** функционально эквивалентного данному обращению фрагмента программы, уже не использующего обращений.

1) ***Избавление от конфликта (коллизии) имен***. В случае совпадения имен формальных параметров и имен локальных объектов с именами глобальных объектов - замени их на новые, уже не несовпадающие.
2) ***Формальная семантика параметров-значений***. Перед телом процедуры проставь операторы присваивания V:=E, V - имя формального параметра, E - соответствующее ему в обращении выражение (фактический параметр-значение)
3) ***Формальная семантика параметров-переменных.*** Замени в тексте процедуры все вхождения имен параметров на соответствующие им при обращении имена переменных (фактический параметр-переменная).

Пример построения модифицированного тела процедуры.

{процедура линейного поиска значения в массиве}

{спецификация: found:=$x \in a \approx \exists\, i \in [1..LenA]\ (x=a_i)$ }

```
procedure poisk(x:T;A:tArray;LenA:tIndex; var found:boolean);
var  i:tIndex;
begin
found:=false; i:=1;
while (i<=LenA) and not found do if A[i]=x then found:=true else inc(i)
end;
```

Построим обращение для poisk(2*i,X,10,b) □ ?

1) (Избавление от коллизии имен)

found:=false; **i1**:=1; while (**i1**<=LenA) and not found do if A[**i1**]=**x1** then found:=true else inc(**i1**)

2) (Присваивание значений параметров-значений)
   x1:=2*i; a:=x;LenA:=10; found:=false; i1:=1; while (i1<=LenA) and not found do if A[i1]=x1 then found:=true else inc(i1)
3) (Замена имен параметров-переменных)
4) x1:=2*i; a:=x;LenA:=10; **b**:=false; i1:=1; while (i1<=LenA) and not found do if A[i1]=x1 then **b:=**true else inc(i1)

т.е. poisk(2*i,X,10,b) □ x1:=2*i; a:=x; LenA:=10; b:=false; i1:=1; while (i1<=LenA) and not found do if A[i1]=x1 then b:=true else inc(i1)

Побочный эффект - фактическое изменение значений глобальных объектов в теле процедуры, не отражаемое формально, т.е. синтаксически, при обращении к процедуре - а потому, *неявное* для программиста - пользователя процедуры.

Пример побочного эффекта. Пусть процедура линейного поиска теперь реализована так

```
procedure poisk(x:T;A:tArray; var found:boolean);
{спецификация: found:=x ∈a ≈ ∃ i∈ [1..LenA] (x=a_i) }
{var  i:tIndex; - сэкономили память!}
begin
found:=false;
while (LenA>=0) and not found do
        if A[i]=x then found:=true else dec(Len)
end;
```

В отличие от параметров процедуры, глобальные объекты никак не участвуют в модификации тела процедуры, потому обращение вида poisk(x,a,b) не только найдет правильное значение b, но и обнулит значение глобальной переменной LenA, содержащую по смыслу фактическую длину массива a и тем самым "закроет" его элементы для дальнейшей обработки.

Потому - либо а) (*правило полной локализации*) вообще *не используй* локально (т.е. в теле процедуры) глобальные объекты, либо, как минимум б) *не изменяй* их значений в теле процедуры. *В любом случае* использования глобальных объектов *помечай* такое использование синтаксически (в виде комментария).

§ 3. Классификация типов данных.

(Начни с определения понятия абстрактного типа данных как области - пары <A,F>, $F \subseteq A \to A$ – см. §1).

Примеры базовой семантики конкретных типов языка Паскаль (в реальности, как мы хорошо знаем, каждый тип обогащен множеством других функций)

Real, integer - арифметика вещественных и целых чисел,

string - полугруппа слов относительно операции конкатенации (сцепления слов),

array[I] of T - множество всех функций $I \to T$, с операцией App применения (аппликации) функции к аргументу, App(a,i)=a[i],

record $n_1:T_1;\ldots;n_m:T_m$ end - именованное декартово произведение (определение см. §1) основных множеств типов $T_1,\ldots,T_m$ с конечным множеством имен $\{n_1,\ldots,n_m\}$, с операцией аппликации, App(r,n)=r.n,

file of T - множество всех последовательностей $N \to T$, с операторами:

reset(f) ≈ маркер(f)=0; режим(f):=чтение;

read(f,x) ≈ inc(маркер(f));x:=f(маркер(f))

область определения: (режим(f)=чтение)& (маркер(f)∈ Dom(f))

rewrite(f) ≈ маркер(f)=0; режим(f):=запись;f:=<> (пустая последовательность)

write(f,e) ≈ inc(маркер(f));f(маркер(f)):=e;

область определения: (режим(f)=запись)

eof(f) ≈ маркер(f)∈ Dom(f) ≈ компонента f(маркер(f)) определена

где маркер(f), режим(f) - некоторые вспомогательные *системные* (т.е. предопределенные, не определяемые пользователем) переменные.

(В отличие от объектного подхода), классический процедурный стиль склоняется к *раздельному* описанию

а) способов определения *основного множества* значений A, при *фиксированном* классе функций F (собственно, и относимых к определению типа) и

б) способов определения класса F - преобразований значений типа (см. § 2).

Это определяет и соответствующую классификацию типов по способу определений значений основного множества.

1) Разделение между ***скалярными и производными*** типами уже проведено в §1. В последнем случае, значения структурированы. Общая схема такой структуры - именованное декартово произведение, т.е. некоторый класс функций. Различие производных типов - в конкретных областях определения и значений этих функций, и (фиксированных для каждого типа) операторов их обработки. Примеры скалярных типов в Паскале - integer, real, char, boolean, перечислимый, ограниченный, производных типов - string, text, array, file, record, set.

2) ***Стандартные и пользовательские*** типы. В первом случае множество значений преопределено, во втором (до)определяется программистом. Примеры стандартных типов - integer, real, char, boolean, string, text; пользовательских типов - перечислимый, ограниченный, array, file, record, set.

3) ***Порядковые*** и непорядковые ***типы***. Понятие перебора (перечисления, пересчета) является одним из наиболее фундаментальных в программировании. Порядковые типы - те множества значений X, на которых определен (хотя бы один) порядок перебора, т.е. сравнение (линейный порядок) $<$, $x<x'$ - "x' следует позже в переборе, чем x". Однако, не *любое* сравнение определяет порядок перебора, а лишь такое, для которого определены (доступны)

- *функция следования* (прямого перебора) succ, succ(x)=min{x': x'>x} и
- *функции предшествования* (*обратного* перебора) pred, pred(x)=max{x’:x’<x}

Примеры порядковых типов Паскаля -

integer - succ(n)=n+1,

char, - succ(c_i)= c_{i+1} (c_i - - i-ый символ в предопределенном языком фиксированном алфавите, таблице символов)

Boolean - false<true

Значения производных типов (array T1 of T2, record, file of T) обычно не считаются порядковыми, но нужный порядок можно (до)определить, если соответствующие базовые типы - порядковые (см. перечисление последовательностей в **§10 и §11**). Очень интересны примеры

- *типа real,* на котором определено обычное сравнение чисел, но из теоретических «мощностных» соображений не может быть определена соответствующая функция следования - не существует наименьшего вещественного числа, большего данного.
- *множества всех бесконечных последовательностей*, с теми же свойствами. (см. "Перечисление последовательностей").

Они *не являются* порядковыми типами.

4) ***Динамические и статические*** производные типы. Значения производных типов - функции. Область их определения может быть фиксирована (статический тип) - пример - множество всех функций [1,n]→T для *фиксированного* n, либо нет (динамический тип), пример - множество всех функций [1,n]→T для *произвольного* n. Примеры статических типов - record, статический array (в стандартном Паскале); динамических - file, string, динамический array (в Object Pascal).

В последнем случае определение типа предполагает явное или неявное наличие соответствующих операторов расширения и сужения области определения. Примеры - оператор Паскаля SetLength явного установления области определения динамических массивов; оператор write увеличивает длину файла неявно.

Практический интерес представляют типы, которые можно назвать *псевдодинамическими.* Это функции, область определения которых не фиксирована, но содержится в некотором фиксированном множестве значений. Пример: множество всех последовательностей f с длиной менее заданной, f:[1..n]→T, n<nMax. В практическом программировании необходимо учитывать конечность памяти - в реальности, все динамические типы являются псевдодинамическими!

5) ***Абстрактные*** типы. Совершенно иной смысл имеет условное разделение абстрактных и конкретных типов. *Абстрактные типы* - типы, точно определяемые как математические объекты (объекты некоторого языка спецификаций), но не как объекты данного языка программирования. Пример - комплексные числа и точки плоскости не есть тип Паскаля, тип record x,y: real end - тип Паскаля. Установление соответствия между (значениями и операциями) абстрактного и конкретного типов называется ***реализацией*** (определением, моделированием или представлением) абстрактного типа.
6) Тесно связано с понятием реализации (как способа определения) деление типов ***по способам доступа.*** На практике, программисту важен именно *способ определения, задания* объекта/функции. Другое дело, что такие определения можно *снова* точно сформулировать в терминах функций/операторов. Существует три важных вида таких определений*:* *рекуррентное (данные последовательного доступа*, пример - файлы и списки, язык описания - функции следования), *явное (данные прямого доступа*, пример - массивы и записи, язык описания - оператор аппликации) и *рекурсивное* (пример - деревья и графы, язык описания - функции на последовательностях)

Пояснить классификацию типов на конкретном примере.

Задача: *найти длину lmax самого длинного слова w* (в текстовом файле f) и само *это слово w*; (слова представлены символьным списком.)

С абстрактной (логической) точки зрения, дана последовательность слов, с точки зрения реализации - динамический тип файл (последовательность символов). Здесь, постановка задачи уже фиксирует реализацию значений - но не операторов! Задача легко решается в терминах абстрактного типа (слова и их длины), основная сложность - реализовать соответствующие операции.

type

{список - абстрактный динамический пользовательский тип последовательного доступа}

slovo=^component;

component=record letter:char;next:slovo end;

{аналогично - найди место в каждой классификации для остальных типов, используемых в спецификации и реализации программы}

procedure P(var f:text; var lmax:integer; var w:slovo);

var v:slovo;

begin

lmax:=0;

w:=ПустоеСлово;

stop:= КончилисьСлова(f); v:=Очередное слово(f); l:= Длина(v);

while not stop do

begin

if l>lMax then begin lmax:=l; Копировать(**w,v**) end;

{подготовка переменных к следующему проходу}

Уничтожить(v);

stop:= КончилисьСлова(f); v:=Очередное слово(f); l:= Длина(v);

end;

Внимание - хитрость примера - обратное влияние или "побочный" эффект реализации - независимое определение функции КончилисьСлова (≈ остаток файла содержит хотя бы одно слово ≈ хотя бы один не пробел) делает практически невозможным (точнее, сильно неэффективным) реализацию процедур ОчередноеСлово и Длина. Отдельные, с точки зрения логики, операторы, могут быть реализованы только совместно.

procedure P(var f:text; var lmax:integer; var w:slovo);

var v:slovo;

```
begin
lmax:=0;
w:=nil; {w:=ПустоеСлово}
ОпределиКончилисьСловаЕслиНетОпределиОчередноеСловоИЕгоДлину
(f,stop,v,l);
{читай файл до непробела/начала слова, если таковое есть - читай все непробелы в список v, одновременно подсчитывая их число в l}
while not stop do
begin
if l>lMax then begin lmax:=l; Копировать(w,v) end;
{подготовка переменных к следующему проходу}
Уничтожить(v);
ОпределиКончилисьСловаЕслиНетОпределиОчередноеСловоИЕгоДлину
(f,stop,v,l);
end;
```

Замечание. Если есть трудности с реализацией порождения, уничтожения и копирования списков - см. "Абстрактные линейные типы данных"

---

§4. Вычисление предикатов.

Предикат (свойство, условие) - произвольная функция с областью определения Boolean={true,false}. Пример (двуместных) предикатов - бинарные отношения, в частности, отношения сравнения <,>,=,<>.

Вычисление предикатов облегчается тем, что существует *общематематический* язык логики (исчисления предикатов) - исключительно выразительный и одновременно компактный - содержащий бинарные операции & (and, в синтаксисе Паскаля), $\vee$ (or), $\neg$ (not) - их относят к операциям алгебры логики, а также два квантора (оператора над множествами) $\forall$ и $\exists$ (отсутствуют в Паскале). Перевод формулы из языка ИП в программу на структурном языке программирование - техническая задача, при учете следующих основных моментов.

1. Написание (спецификация) формул может быть само по себе достаточно сложной задачей. Математическая логика располагает множеством стратегий таких спецификаций в виде алгоритмов записи любого предиката в некоторой ***канонической (нормальной) форме.*** Наиболее простые из таких форм для бескванторных формул - ***днф*** (дизъюнкция элементарных коньюнкций), ***кнф*** (конъюнкция элементарных дизъюнкций); по определению, *элементарные* формулы могут содержать отрицание, но не знаки бинарных логических операций.
2. Для кванторных формул канонической является т.н. *предваренная нормальная форма* вида $Q_1x_1\ldots Q_n x_n B(x_1,\ldots,x_n,z)$ (где Q_i - либо $\exists$ либо $\forall$, причем каждые два соседних квантора можно считать различными, а B уже не содержит кванторов).

Пример. Определить принадлежность точки p(x,y) к фигуре, составленной из 4 колец. Каждое кольцо задаваемется центром и длинами внутреннего и внешнего радиусов.

Решение - стратегия днф.

а) Точка принадлежит к фигуре, если она принадлежит одному из колец.

type

```
tPoint=record x,y:real end;
tRing=?
tFigure=array[1..n] of tRing;

function ПринадлежитФигуре(p: tPoint; figure:tFigure):boolean;
begin
ПринадлежитФигуре:=
ПринадлежитКольцу(p,figure[1]) or
        ПринадлежитКольцу(p,figure[2]) or
        ПринадлежитКольцу(p,figure[3])
end
```

b) Точка принадлежит кольцу, если она находится одновременно внутри внешнего и вне внутреннего окружностей кольца.

```
type
tRing =record center:tPoint; radius1,radius2:real end;
function ПринадлежитКольцу(p:tPoint; Ring:tRing):boolean;
begin
ПринадлежитКольцу:=
ПринадлежитОкружности(p,Ring.center,Ring.radius1) and
not ПринадлежитОкружности(p,Ring.center,Ring.radius2)
end;
```

d) Принадлежность точки P с заданными координатами Px,Py к окружности O с центром Ox,Oy и радиусом Oradius описывается атомарной формулой

$(Px\text{-}Ox)^2+(Py\text{-}Oy)^2 \leq Oradius^2$

```
function
ПринадлежитОкружности(p:tPoint,center:tPoint; radius:real):boolean;
begin
ПринадлежитОкружности:=
        sqr(P.x-Center.x)+sqr(P.y-Center.y)<=sqr(radius)
end;
```

3. ***Алгоритмическое определение операций алгебры логики***.

a) y:=not b ≈ if b=true then y:=false else y:=true
b) y:=b1 and b2 ≈ 1) y:=false; if b1=true then if b2=true then y:=true; ≈ 2) y:=true; if b1=false then y:=false; if b2=false then y:=false;
c) y:=b1 or b2 ≈ 1) y:=true; if b1=false then if b2=false then y:=false 2) y:=false; if b1=true then y:=true; if b2=true then y:=true

Внимание - указанные эквивалентности верны лишь для *определенных* значений b1,b2. Программы 1) определенные значения y при неопределенном значении b2, программы 2) - нет. Семантика 1) соответствует т.н. ***быстрому*** означиванию операций и определяют т.н. *условные коньюнкцию и дизъюнкции* - строго говоря, отличных от ***классических*** (семантика 2). В Паскале эта неоднозначность ("темное" место языка) разрешается директивой компилятора $B. Разные значения директивы может сделать одну и ту же программу верной или неверной!

Задача (линейный поиск)

```
type
tIndex=1..MaxLen;
ArrayOfT=array[tIndex] of T;
procedure Poisk(x:T; a:ArrayOfT; LenA:tIndex; var found:boolean; var index:tIndex);
begin
index:=1;
while (index<=LenA) and (x<>a[index]) do inc(index);
found:= index<=LenA
end;
```

При быстром означивании формула (index<=LenA) and (x<>a[index]) при index=LenA+1 дает false (программа верна), при классическом - значение неопределено (программа неверна). Совет: желательно избегать неопределенных значений -

```
index:=1; found:=false;
while (index<=LenA) and not found do if x=a[index] then found:=true else inc(index);
```

4. ***Вычисление кванторных формул.***

a) y:=∃ x∈ X (B(x))

Рассмотрим частный (для классической логики), но наиболее важный для программирования случай. X - множество значений порядкового динамического типа, для которого определены функция следования (перебора) и предикат конца перебора значений - заданный, например, в виде сравнения. Тогда $\exists\ x\in X\ (B(x))$ - это кратная дизъюнкция и y равно true, если в реккурентной последовательности y_0=false, $y_{i+1}= y_i$ or $B(x_i)$ найдется значение true - либо false, если такового нет. Поэтому, можно воспользоваться общей схемой вычисления членов реккурентных последовательностей:

y:= $\exists\ x\in X\ (B(x)) \approx$

y:=false; i:=1; while (i<=LenX) and not y do if $B(x_i)$ then y:=true else inc(i)

Точно также можно определить формулу $\forall\ x\in X\ (B(x))$ как кратную конъюнкцию:

y:= $\forall\ x\in X\ (B(x)) \approx$

y:=true; i:=1; while (i<=LenX) and y do if $B(x_i)$ then y:=false else inc(i)

Пример вычисления кванторной формулы. Последовательность A длины LenA (строго) периодическая (или, попросту - кратная), если оно состоит из двух или более одинаковых "сцепленных" подпоследовательностей некоторой длины k, называемой в этом случае периодом A. Выяснить, является ли данная последовательность строго периодической.

Спецификация

b:=A - строго периодическая ≈

b:=$\exists$ k $\in$ [1,LenA div 2] (k - период A) ≈

b:=$\exists$ k $\in$ [1,LenA div 2] (LenA mod k =0 → $\forall$ i$\in$ [1..LenA-k] (A[i]=A[i+k]))

```
type
tIndex:1..MaxLen;
tArray:array[tIndex] of T;

function Period(k:tIndex; a:tArray; LenA:tIndex):boolean;
var  b2:boolean; UpperLimit2:tIndex; {= LenA-k }
```

```
begin
b2:=LenA mod k; UpperLimit2:= LenA-k; i:=1;
while (i<= UpperLimit2) and b do
if A[i]<>A[i+k] then b2:=false else inc(i)
Period:=b2
while


end;
function Periodic(a:tArray; LenA:tIndex):boolean;
var UpperLimit1:tIndex; {=LenA div 2} b1:boolean;
begin
b:=false; k:=1; UpperLimit:=LenA div 2;
while (k<=UpperLimit) and not b do
if period(k,A.LenA) then b:=true else inc(k);
Periodic:=b;
end;
```

§5. Поиск.

Основной интуитивный смысл структур данных - "хранилище" данных, обеспечивающее операции доступа (чтения) и записи, что, при формальном определении этих структур как функций соответствует вычислению следующих основных операторов

- аппликация (по хранилищу f и указателю/имени x определить значение f(x)),
- (пере)определению значения функции f в точке x новым значением y, и
- (для динамических типов) расширению/сужению области определения функции.

Понятие хранилища данных тесно связано по смыслу, но существенно шире по объему понятия структуры (производного типа). Так, в качестве хранилища, при фиксации кодирования, могут выступать и скалярные значения.

Так, например, по основной теореме арифметики, любое $n, n \in N$, есть некоторое произведение степеней простых чисел $\Pi\, p(i)^{a(i)}$ (p(i) i-е простое число). Следовательно, его можно рассматривать как хранилище последовательности a, т.е. потенциальную или "виртуальную", т.е. воображаемую структуру.

Важно уточнить, что фактически может храниться не сама функция (в табличном виде, пример - данные *прямого* доступа - массивы и записи), но способ ее определения - реккурентный (данные *последовательного* доступа, пример - файлы) или рекурсивный (пример - деревья)

Задача поиска формулируется в двух внешне различных формулировках.

a) $b := y \in Val(f) \approx \exists\, x \in Dom(f)\ (f(x)=y)$ - выяснить, храниться ли данное значение y в структуре f.

b) x:=min{x': f(x')=y} определить первое (в некотором порядке) "имя", под которым храниться данное значение.

(задача - обратная к задаче доступа).

В реальности, эта одна задача. С конструктивной точки зрения, мы не не можем ответить на вопрос a), не предъявив такое x, что f(x)=y и не можем найти x в b), не выяснив, существует ли таковое вообще.

$b, x := y \in Val(f) \approx \exists\, x \in Dom(f)\ (f(x)=y),\ \min\{x' \in Dom(f): f(x')=y\}$

К задаче поиска легко сводяться многие важные задачи

- выяснить, храниться ли в структуре f значение с заданным свойством, в частности,
- выяснить, храниться ли в структуре f значение *почти равное* заданному (т.е. хорошее приближение заданного значения)
- определить *все* "имена", под которым храниться данное значение (значение с заданным свойством) и т.п.

В любом случае, решение предполагает наличия некоторого порядка перебора на $Dom(f)=\{x_1,..,x_n,..\}$.Тривиальное решение задачи - ***линейный поиск*** - мгновенно следует из схемы вычисления ∃-свойств (см. "Вычисление свойств") в данном случае основанном на реккурентном отношении:

(*) $\exists\, x \in \{x_1,..,x_n,..\}\ (f(x)=y) \approx (f(x_1)=y)$ or $\exists\, x \in \{x_2,..,x_n,..\}\ (f(x)=y))$

```
procedure LinearSearch(f:tStructure;y:tVal;var b:boolean;var x:tDom);
begin
b:=false; i:=1;
while (i<=LenDom) and not b do if f(x_i)=y then begin b:=true;x:=x_i end else
inc(i)
end;
```

Понятно, это схема. Пересчет tDom и условие его окончания может выглядеть совершенно иначе.

Пример (поиск в конечном дереве и графе). Дерево (граф) в качестве хранилища - функция вида $A^* \to T$, A - некоторый алфавит, A^* - множество всех конечных последовательностей (ветвей дерева), а T - тип вершин дерева (графа). Для решения задачи поиска достаточно определить некоторый порядок перебора ветвей.

```
procedure LinearSearch(f:tTree;y:tVal;var b:boolean;var x:tDom);
begin
b:=false; x:=ПервоеСлово; {естественно взять в качестве первого пустое слово - других может и не быть!}
while not КончилисьСлова(f) and not b do
if f(x)=y then begin b:=true else x:=Следующее(x)
end;
```

Окончание решения - см. "Перечисление последовательностей".

Очевидно, в худшем случае алгоритм линейного поиска требует полного перебора всех элементов tDom. Можно ли ускорить поиск? Да, если предполагать наличие некоторого сравнения (линейного порядка) < на tDom (в отличие от tDom, это не обязательно порядок перебора) и рассматривать в качестве хранилищ *упорядоченные* последовательности (или, в общем случае, *монотонные* функции) f: tDom$\to$tVal

$$\forall\ x,x' \in tDom\ (x \le x' \to f(x) \le .f(x'))$$

Ограниченный поиск. Первая идея "сокращения пространства поиска" основана на простом наблюдении - в случае монотонности f бесполезно проверять равенство f(x)=y для тех x, для которых f(x)>y :

$\exists\ x \in tDom\ (f(x)=y) \approx \exists\ x \in \{x_1,\ldots, x_k\}\ (f(x)=y)$, где $k=\min\ \{k': f(x_k) \ge y\}$.

```
{предусловие - f монотонна (упорядочена)}
procedure RestrictedSearch(f:tOrderedStructure;y:tVal;var b:boolean;var x:tDom);
```

```
var UpperLimitFound:boolean; {∃ x ∈ tDom (f(x) ≥ y)}; y1:tVal;
begin
{найди первое x, что f(x) ≥ y}
UpperLimitFound:=false; i:=1;
while (i<=LenDom) and not UpperLimitFound do
begin y1:=f(x_i); if y1>=y then begin UpperLimitFound:=true;x:=x_i end else inc(i); end;
if UpperLimitFound then b:=y1=y else b:=false
end;
```

Пример применения схемы - поиск в упорядоченном файле

```
{предусловие - f монотонна (упорядочена)}
procedure RestrictedSearch(f:tOrderedFile;y:tVal;var b:boolean;var n:Cardinal);
var UpperLimitFound:boolean; {∃ x ∈ tDom (f(x) ≥ y)}; i:Cardinal;
begin
{найди первое x, что f(x) ≥ y}
UpperLimitFound:=false;
reset(f); {j:=1} i:=0;
while not eof(f) {(i<=LenDom)} and not UpperLimitFound do
begin read(f,y1); {y1:=f(j);inc(j)}
if y1 >=y then begin UpperLimitFound:=true;n:=i end else inc(i);
end;
if UpperLimitFound then b:=y1=y else b:=false
end;
```

Идея ***дихотомического поиска*** (поиска методом деления пополам) также предполагает упорядоченность f и основана на соотношении

(*) $\exists\, x \in \{x_{n1},..,x_{n2}\}\ (f(x)=y) \approx$

$\exists\, x \in \{x_{n1},..,x_m\}\ (f(x)=y))$ xor $\exists\, x \in \{x_{m+1},..,x_{n2}\}\ (f(x)=y))$, где m=(n1+n2) div 2 ≈

(для монотонной f)

$(f(x_m)=y)$ xor

$(f(x_m)<y)\ \&\ \exists\, x \in \{x_{n1},..,x_{m-1}\}\ (f(x)=y))$ xor

$(f(x_m)>y)\ \&\ \exists\, x \in \{x_{m+1},..,x_{n2}\}\ (f(x)=y))$

(т.е. в реальности делим Dom(f) на 3 части, не на 2)

```
{предусловие - f монотонна (упорядочена)}
procedure Dichotomy(f:tOrderedStructure;y:tVal;var b:boolean;var x:tDom);
UpperLimit, LowerLimit:tDom; {верхняя и нижняя границы пространства поиска}
begin
b:=false; UpperLimit:=1; LowerLimit:=LenDom;
while (UpperLimit>LowerLimit) and not b do
begin
m:= (UpperLimit+LowerLimit) div 2;
if f(x_m)=y
then b:=true
else if f(x_m)<y  then LowerLimit:=m else UpperLimit:=m
end; {while}
end; { procedure Dichotomy }
```

Традиционно дихотомический поиск считается *быстрым* поиском, поскольку позволяет найти ответ не более, чем за $\ln_2 n$ сравнений, в то время как верхняя оценка линейного поиска - n сравнений (где n=Card(Dom) - число элементов в пространстве поиска Dom). Однако, нельзя забывать, что, в отличие от линейного поиска, этот алгоритм требует вычисления $f(x_m)$, которое может оказаться неоправданно дорогим - например, в случае поиска в файле.

Идея дихотомии (как и ограниченного поиска!) непосредственно продолжается на деревья (и графы) специального вида - *деревья поиска*. Дерево в качестве хранилища есть некая функция f, f:A*→T, где A* - множество путей (ветвей) дерева, а T - содержимое его вершин. Разберем самый простой случай бинарных деревьев - A={0<1} (кратное обобщение на случай произвольного конечного $A=\{a_1<\ldots< a_n\}$ - техническая задача).

Функция f, f:{0<1}*→<T,$<_T$ > - ***дерево поиска***, если оно f монотонна относительно *лексикографического* порядка << на словах (см. "Перечисление последовательностей") : $\forall$ v1,v2 $\in$ {0,1}* (v1<<v2 → f(v1)$<_T$ f(v2))

```
procedure OrderedTreeSearch(f:tOrderedTree; y:tVal; var b:boolean; var x:tDom);
begin
b:=false; v:=ПустоеСлово;
while (v ∈ Dom) and not b do
begin
if f(v)=y then b:=true
else if f(v)>y then v:=v⊕0 else v:=v⊕1;
end;
end;
```

Здесь ⊕ - операция приписывания (конкатенации) буквы к слову. В случае реализации деревьев ссылками (см. обозначения в "Нелинейные типы данных"):

```
v:=ПустоеСлово  ≈ p:=Root;
v:=v⊕0          ≈ p:=p^.left
v:=v⊕1          ≈ p:=p^.right
v∈ Dom          ≈ p<>nil
```

где Root, p - ссылки на корень и текущую вершину дерева.

§ 6. Упорядоченные типы.

Термин “(структурный) упорядоченный тип” по сути отсутствует в современных языках программирования как языковое понятие. Однако, логическая естественность определения и практическая польза часто вынуждает рассматривать упорядоченные последовательности (и монотонные функции, в целом) как абстрактный тип данных.

Естественные операции, призванные сохранять упорядоченность – вставка и удаление компонент, объединение (слияние), пересечение, разность последовательностей имеют явные теоретико-множественные корни:

C=Вставка(A,b) $\approx \forall\, i \in Dom(C)\ (c_i \in A$ or $c_i = b)$ & Упорядочена(C)

C=Удаление(A,b) $\approx \forall\, i \in Dom(C)\ (c_i \in A$ and $c_i \neq b)$ & Упорядочена(C)

$C = A \cup B \approx \forall\, i \in Dom(C)\ (c_i \in A$ or $c_i \in B)$ & Упорядочена(C)

$C = A \cap B \approx \forall\, i \in Dom(C)\ (c_i \in A$ and $c_i \in B)$ & Упорядочена(C)

$C = A \setminus B \approx \forall\, i \in Dom(C)\ (c_i \in A$ and $c_i \notin B)$ & Упорядочена(C)

где $x \in f \approx \exists\, i \in Dom(f)\ (x = f(i))$

Практическая мотивация их использования - в отличие от алгоритмов сортировки (см. след. тему), все следующие ниже алгоритмы *однопроходные,* т.е. *сохранять* упорядоченность легче, чем сортировать (особенно – для типов с последовательным доступом!)

Направляющая общая идея реализации - кратный *ограниченный поиск* в упорядоченной последовательности - для ответа на вопрос $x \in a$, достаточно найти “барьер”, т.е. первый a_i такой, что $x \le a_i$. (Подробнее см. "Поиск").

1. Разность упорядоченных файлов. Здесь и далее TInfo – произвольный тип, на котором определен некоторый линейный порядок <.

```
Type
TFile=file of tInfo;
{предусловие – удобнее считать, что файл B не пуст}
procedure Разность(      var A,B:tFile;
                 var C:tFile); {c:=a\b}
var
cA,cB:tInfo;{текущие элементы последовательностей}
BarrierFound, {= ∃i (cA<= bi)}
found:boolean; {cA∈ b}
begin
reset(A); reset(B);rewrite(C);
read(B,cB); {по условию файл B не пуст }
{ в противном случае, решение очевидно – скопировать A в C}
while not eof(A) do
begin
read(A,cA); BarrierFound:= cA<=cB ;
while not eof(B) and not BarrierFound do
begin
read(B,cB); if cA<=cB then BarrierFound:=true
end;
found:=false; if BarrierFound then if cA=cB then found:=true;
if not found then write(C,cA)
end;
end; { Разность }
```

2. Слияние массивов

```
type
tIndex=1..MaxIndex;
tArray=array[tIndex] of tInfo;
```

```
procedure Слияние(A,B:tArray;LenA,LenB:tIndex;var C:tArray;var LenC:tIndex); {c:=a∪b}
var
i,j:tIndex;
BarrierFound: boolean; {= B[j]≤A[i] }
begin
LenC:=0; j:=1;
for i:=1 to LenA do
begin
{каждый  элемент A[i] должен попасть в C, но до этого}
{скопируй все элементы B[j], B[j]≤A[i] в C }
BarrierFound :=false;
while (j<=LenB) and not BarrierFound do
begin
if B[j]≤A[i]
then begin C[LenC]:=B[j];inc(LenC);inc(j) end
else BarrierFound :=true;
end;
C[LenC]:=A[i];inc(LenC)
end; {while}
end {procedure Слияние }
```

3. Вставка в упорядоченный список

```
Type
{определение списка}
pList=^tList;
tList=record
info:tInfo; {некий тип со сравнением <}
next:pList
    end;
```

```
Procedure Вставка(        var pA:pList;
{A - исходная упорядоченная последовательность с барьером}
                pX:pList; {ссылка на вставляемое значение x} );
var
        x:tInfo;
This,Prev, {ссылки на текущую и предыдущую компоненты списка}
Begin
        x:=pX^.info;
        {найди ссылку This на первую компоненту со значением info, не
меньшим x}
        Prev:=nil; This:=pA; found:=false
        While (This<>nil) and not found do
        If This^.info>=x
        then found:=true
        else begin Prev:=This;This:=This^.next end;
        {found=true и This<>nil, по определению барьера}
        {вставляем между Prev и This}
        pX^.next:=This; if Prev<>nil then Prev^.next:=pX
End;
```

§ 7. Сортировка.

Сортировка - преобразование данной последовательности x в упорядоченную (далее, по умолчанию ≈ монотонно неубывающую) последовательность, содержащую те же компоненты. Чуть более формальная спецификация **-**

Предусловие: VAL(x)=X∈ N→T

Постусловие:

VAL(x)=X'∈ N→T & X' монотонна & X' – перестановка X

Мотивация - упорядоченность последовательностей обеспечивает более быстрый поиск (см. "Поиск").

Все излагаемые ниже алгоритмы сортировки основываются на простом факте: если F - некая процедура, применение которой к массиву увеличивает длину отсортированной части массива (скажем, максимального упорядоченного начального его фрагмента), то кратное применение F обязательно упорядочит весь массив.

Задача - найти такой оператор F - по возможности, с достаточно быстро растущим N, N=N(a). Наиболее простые для понимания идеи нахождения нужного F основаны непосредственно на определении (спецификации) упорядоченности.

Последовательность a ∈ [1..n]→T упорядочена

≈ a) ∀ i ∈ [1..n-1] ($a_i \le a_{i+1}$) (идея - ***сортировка простым обменом*** - обмен значений "неправильных" соседних пар)

≈ b) $\forall$ i $\in$ [1..n-1] $\forall$ j $\in$ [i+1..n] ($a_i \le a_j$) (идея ***- "пузырьковая" сортировка*** - обмен значений "неправильных" пар нужного вида) ≈ c) $\forall$ i $\in$ [1..n-1] (a_i = min (a[i,n]) (идея - поиск соответствующего минимума)

{здесь и далее tIndex=1..MaxLen; tArray=array[tIndex] of T, для некоторого типа T с некоторым определенным на его элементах сравнением <, LenA - фактическая длина массива A};

```
procedure СортировкаПростымОбменом (var A: tArray; LenA:tIndex);
var ПустойПроход:boolean; {=∀ i ∈ [1..n-1] (ai ≤ ai+1) }
begin
repeat
    ПустойПроход:=true;
    for i:=1 to n-1 do
    if a[i]>a[i+1] then begin Обмен(a[i],a[i+1]); ПустойПроход:=false end;
until ПустойПроход
end; { СортировкаПростымОбменом }
```

Замечание о сходимости алгоритма. 1 проход *не* упорядочивает массив, *но* - увеличивает отсортированную часть!

Предыдущая тема дает еще одну вариацию начальной идеи сортировки - если операция F сохраняет упорядоченность и увеличивает длину массива, то ее кратное применение упорядочивает массив. К таким, очевидно, относятся операции вставки и слияния.

```
procedure СортировкаВставкой(var A:tArray;LenA:tIndex);
var i:tIndex;
begin
for i:=1 to LenA do {Вставить A[i] в упорядоченный к этому времени массив A[1..i-1]}
end;
```

```
{для простоты, положим LenA=2^n , n∈N}
procedure СортировкаСлиянием(var A:tArray;LenA:tIndex);
var
i,
l,              {=2^i , длина сливаемых подмассивов}
m : tIndex;     {=2^(n-i) число слияний}

begin
l:=1; m:=LenA;
for i:=1 to LenA div 2 do
{Слить попарно все упорядоченные к этому времени подмассивы длины 
2^i , всего 2^(n-i) слияний }
t:=1; for k:=1 to m do
begin
    {k-е слияние - непосредственно слить A[t .. t+l-1], A[t+l .. t+2*l] в A[t
.. t+2*l] проблематично, потому сначала сливаем A[t .. t+l-1], A[t+l .. t+2*l]
в B[t .. t+2*l] а затем копируем B[t .. t+2*l] в A[t,t+2*l]} t:=t+l
end; { for k:=1 to m }
A:=B;l:=l*2;m:=m div 2;
end; { i:=1 to LenA div 2 }
end; { procedure СортировкаСлиянием }
```

§8. Машинно-ориентированное программирование.

Как уже отмечалось в §1, универсальным языком определения всевозможных структур управления (равно как и структур данных - см. "Абстрактные линейные типы" и "Абстрактные нелинейные типы") является язык ссылок - язык произвольных блок-схем или ссылки на оператор в виде оператора goto. Этот язык крайне неудобен, с точки зрения обычной человеческой логики в силу своей примитивности, т.е. принципиальной неструктурированности. Но *именно поэтому* он удобен для технической реализации, и только поэтому компьютеры возможны как физические устройства.

Реализация программ языков программирования на таком чисто ссылочном языке низкого уровня называется ***трансляцией*** (от англ. translation - перевод). Важно, что такой перевод может осуществляться *автоматически,* с помощью алгоритмов формальной текстовой обработки. Трансляция программ - непременный предварительный этап перед их исполнением - в реальности могут исполняться только программы низкого уровня.

Трансляция и последующее исполнение может осуществляться либо над программой в целом (компиляция), либо пошагово - "пооператорно" (интерпретация; при этом программа рассматривается как композиция последовательности операторов).

Написание программ-трансляторов - сложная задача. Наметим лишь несколько базовых идей трансляции, используя **модельный язык низкого уровня** (подъязык языка Паскаль), определяемого следующим образом.

Операторы (или, в терминологии языков низкого уровня - команды).

1. ***Бинарное присваивание*** вида v:=[v•]v', где v -переменная, v' - переменная или константа. Семантика - обычная, но с очевидным оттенком "довычисления"
2. ***Составной оператор*** **;** (в машинных языках обычно никак не обозначается)
3. ***Оператор условного перехода*** вида [if B then] goto M, где B элементарная логическая формула (т.е. формула, не содержащая логических операций) - например, сравнение. Содержательная семантика - "после этого оператора будет выполняться не следующий за ним, но тот, перед которым стоит метка вида M: "

Формальная семантика этого оператора довольно сложна - в кажущемся противоречии с определением оператора, этот оператор как будто не меняет значения ни одной переменной (на деле, он не меняет значений *пользовательских* переменных). Именно поэтому оператором goto не рекомендуют использовать в обычной программистской практике. Больше того, реальность еще сложнее - машинные языки содержат операторы перехода на *переменные-метки* вида goto x^ - перейди на оператор, метка на который содержится в переменной x.

Нетрудно доказать (индукцией по длине выражения E), что любой традиционный оператор присваивания выражается (транслируется) в терминах бинарного присваивания и составного оператора. Также несложно показать, что любая структура управления (блок-схема) выражается в терминах составного оператора и условного перехода. Проще сделать это для структурных блок-схем.

if B then S1 else S2 ≈ if B then goto M1; S2; goto M2; M1:S1; M2:

while B do S ≈ M1: if B=false then goto M2; goto M1; M2:

Наш язык также содержит *единственный* тип данных - **битовая строка** (двоичное поле)- последовательность символов '0' и '1', с операцией указания подполя x[n,l] - подстрока длины l, начинающаяся с n-го символа строки x (n,l - целые переменные)

Этот тип является *универсальным* в том смысле, что значение любого другого скалярного типа можно представить (моделировать - причем, разными способами!) в виде битовой строки. Способ такого представления называется *форматом (типа) данных.*

Например, любому символу некоторого алфавита Char можно сопоставить некоторую битовую строку - скажем, i-ому символу алфавита сопоставить i-ю битовую строку в словарном порядке строк (подходящей длины n).

Натуральным числам можно сопоставить

a) их запись в *двоичной системе счисления* (*двоичный формат*). Алгоритмы перевода в k-ичную систему счисления, $k \in Z$, см. "Задачи текстовой обработки" или
b) их запись в обычной 10-ной системе счисления (предварительно перекодировав алфавит {'0'..'9'} битовыми строками) (*символьный формат*)

Этот способ "работает" для любых типов данных - значения любого типа можно как-то выразить в некоторой системе обозначений! Далее для простоты мы полагаем, что работаем лишь с одним выбранным форматом каждого скалярного типа - их представлением в виде строк некоторой фиксированной длины Len(T).

Все алгоритмы вычисления соответствующих операций над моделируемыми типами данных представляются в виде алгоритмов формальной обработки битовых строк. (см. пример - реализацию сложения "столбиком" в "Задачи текстовой обработки")

Реализация структурных типов.

Наличие в нашем языке операции определения подстроки позволяет сделать неожиданный вывод. Достаточно иметь в нашем языке *единственную* переменную - с некоторым стандартным именем Memory (память) - указывая все нужные значения как некоторые подстроки Memory[i,l] этой переменной. Далее (когда можно) мы используем обычные имена переменных (идентификаторы) вместо непривычных имен вида Memory[i,l]. И все же упомянутый факт принципиален для реализации структурных типов.

Индекс i (номер позиции начала поля) называется *адресом* значения Memory[i,l]. Понятно, что когда длина поля l заранее известна (как, например, в условленном нами случае представления скалярных типов), то по значению i можно однозначно восстановить содержимое поля Memory[i,l]. В этом смысле адрес i является указателем (индексом, ссылкой на, "именем") значения Memory[i,l]. Таким образом, каждое имя переменной имеет числовое значение - адрес начала поля соответствующей этой переменной.

Реальная ситуация чуть сложнее - нумеруются не двоичные разряды (биты), но их группы (двоичные слова, в терминологии машинных языков) - например, байты (группы из 8 битов). Впрочем (поскольку каждое значение занимает целое число слов) это - явно не принципиальный в наших рассуждениях момент.

Нетрудно представить *значения* структурных типов в виде совокупности полей. Так, массив array[1..n] of T - это последовательность n битовых полей, каждое длины Len(T) - т.е. некоторое битовое поле Memory[AdrA,L] длины L=Len(T)*n

Таким образом, Memory[AdrA+(i-1),Len(T)] - подполе, соответствующее значению a[i]. Строго говоря, синтаксис операции указания подполя не позволяет использовать аргументы-выражения, но соответствующее значение

нетрудно вычислить и таким образом реализовать основную для массивов *операцию* выборки (доступа) a[i].

Аналогично, запись record N_1:T_1;…; N_m:T_m end - последовательность m битовых полей разных длин Len(T_1),…,Len(T_m) - т.е. снова некоторое битовое поле Memory[AdrRec,L] длины L= Len(T_1)+…+Len(T_m). Нетрудно вычислить AdrRec+ Len(T_1)+…+Len(T_{k-1}) - адрес начала поля, содержащего значение r. N_k и таким образом реализовать операцию доступа по имени, для каждого имени N_k.

Закончим тему примером трансляции "вручную".

Пример. Написать программу на определенном выше языке, определяющую, есть ли данное значение x в последовательности a из 10 чисел, если известны адрес AdrX числа x и AdrA начала поля, содержащего числа из a. Все числа представлены в некотором формате длины L=16.

Писать программу непосредственно на языке низкого уровня - мало вдохновляющее работа. Потому, оттранслируем готовую программу на языке высокого уровня.

```
b:=false;
i:=1;
while (i<=n) and not b do
if a[i]=x then b:=true else i:=i+1
```

Проведем трансляцию в несколько этапов - упрощающих трансляцию для нас (не для компьютера, а потому не характерных для алгоритмов автоматической трансляции).

a) Избавимся от операций выборки.

b:=false;

i:=1;

CurrentAdr:=AdrA; {адрес начала текущей компоненты}

while (i<=n) and not b do

if Memory[CurrentAdr,16]=x

then b:=true

else begin i:=i+1; **CurrentAdr:=CurrentAdr+16** end;

b) Избавимся от сложных выражений

b:=false;

i:=1;

CurrentAdr:=AdrA;

{go:= (i<=n) and not b}

{предполагаем быстрое означивание -см. "Вычисление свойств"}

go:=false; if i<=n then if b=false then go:=true

while go do

begin

if a[i]=x then b:=true else begin i:=i+1; CurrentAdr:=CurrentAdr+16 end;

go:=false; if i<=n then if b=false then go:=true

end;

c) Избавимся от структурных операторов.

b:=false;

i:=1;

CurrentAdr:=AdrA;

go:=false;

{if i<=n then if b=false then go:=true }

if i<=n then goto M1

if b=false then goto M1

go:=true;

M1: **if go=false then goto M2** {while go }

{if a[i]=x then b:=true else inc(i);}

if a[i]=x then goto M3

i:=i+1;

CurrentAdr:=CurrentAdr+16

goto M4

M3: b:=true

M4: {go:=false; if i<=n then if b=false then go:=true}

if i<=n then goto M5

if b=false then goto M5

go:=true;

M5: goto M1

Окончание - надо заменить идентификаторы на указание адреса x→Memory[AdrR,16] и соответствующие выражения для остальных переменных.

§ 9. Абстрактные линейные типы.

Напомним, понятие абстрактного типа *относительно* - это те типы, существующие как математическое понятие, т.е. пара <A,F>, $F \subseteq A \to A$ (см. "Основные понятия"), которые отсутствуют в данном языке программирования. Тип, абстрактный для одного языка, может быть вполне конкретным для другого.

Рассмотрим некоторые общие принципы реализации абстрактных типов на примере двух простых и естественных типов - ***стек и очередь***. (Применение - см. далее "Перечисление последовательностей" и "Нелинейные абстрактные типы данных").

В обоих случаях, ***основным множеством типа*** является множество всех (конечных) последовательностей f *произвольной* длины LenF над некоторым базовым типом T - $f \in N \to T$, Dom(f)=[1..LenF], $LenF \in N$. Т.е. оба типа, вообще говоря, - *динамические*; в случае фиксированной длины можно говорить о *статических,* (или, чуть точнее - *псевдодинамических* стеках и очередях, см. "Классификация типов").

Интуитивная "идея" стека - "тот, кто пришел последним, уходит первым" (**L**ast **I**n **F**irst **O**ut, англ.) или идея магазинной памяти (здесь имеется в виду магазин, т.е. обойма автоматического оружия), что можно точно (=формально) описать следующим образом.

Операторы и функции определения стека.

Операторы.

Create(r) - создать пустой стек с именем r. Начиная работать в любом *состоянии* s, оператор доопределяет s, добавляя в него пару r→<> (<> - пустая последовательность) (см. определение состояния в "Основные понятия")

Destroy(r) - обратная к Create операция - убирает из текущего состояния пару r→R. Имя r перестает существовать, доступ к стеку теряется.

Внимание - важно понимать, что эти операторы добавляют и удаляют *имена,* не значения - т.е. изменяют область определения состояния Dom(s). Это новый для нас случай *динамического распределения* памяти (до сих пор неявно считалось, что Dom(s) определяется статически - *до* выполнения операторов программы, в области определения переменных var)

Push(r,x) - добавить компоненту x (базового типа T) в верхушку (конец) стека - начиная работать в состоянии r→R, со стеком длины Len(R) Dom(R)=[1..Len(R)], этот оператор неявно доопределяет Dom(R) до [1..Len(R)+1] и осуществляет присваивание R[Len(R)+1]:=x.

Pop(r,x) - осуществляет присваивание x:=R[Len(R)] (кладет верхнюю компоненту стека в x) и *сокращает* Dom(R) до [1..Len(R)-1] (*уничтожает* верхушку стека).

Функция (предикат).

Empty(r)=true ≈ Len(R)=0 (т.е. стек пуст)

Пример. Обратить (перевернуть) содержимое символьного файла «небольшой длины».

```
procedure Revolution( var InputFile, OutputDile: text);
var
c: char; {текущий символ}
r:tStack; {символьный стек}
begin
```

```
create(r); reset(InputFile);
while not eof(InputFile) do begin read(InputFile,c);push(r,c) end
close(InputFile);
rewrite(OutputFile);
while not Empty(r) do begin pop(r,c); write(OutputFile,c) end;
close(OutputFile);
destroy(r)
end;
```

Интуитивная идея очереди - "тот, кто пришел первым, первым и уйдет" (**F**irst **I**n **F**irst **O**ut, англ.)

Операторы и функции определения очереди.

Операторы.

Create(r), Destroy(r) - тождественны соответствующим стековым операциям.

Put(r,x) - ***Поставить В Очередь -*** добавить компоненту x (базового типа T) в *конец* очереди - начиная работать в состоянии r→R, со очередью длины Len(R) Dom(R)=[n..m], этот оператор неявно доопределяет Dom(R) до [n..m+1] и осуществляет присваивание R[m]:=x.

Get(r,x) - ***Вывести Из Очереди -*** осуществляет присваивание x:=R[n] (кладет *первую* компоненту очереди в x) и *сокращает* Dom(R) до [n+1..m] (*уничтожает* начало очереди).

Предикат Empty(r) (очередь пуста) - тождественен соответствующему предикату над стеком.

Стек и очередь - абстрактные для Паскаля типы, потому их необходимо реализовать имеющимися в нем средствами.

Реализация стеков и очередей (псевдодинамическими) массивами.

Псевдодинамические массивы - последовательности *переменной* длины m, m≤MaxLen, где MaxLen - *константа.*

const MaxLen=100;

type

tIndex=1..MaxLen;

tArray=array[tIndex];

tPseudoArray=

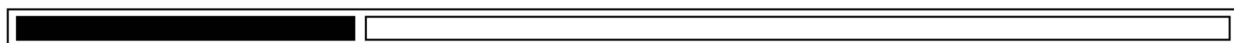
record content:tArray; {содержимое/компоненты массива}

{можно задать len:tIndex; фактическая длина массива}

{или - принимаемый далее вариант}

top:tIndex; {len+1, первое свободная позиция в массиве, начало ***кучи*** - незаполненной части массива }

end;



Нетрудно сопоставить содержимому стеков содержимое массивов, а стековым операциям - соответствующие алгоритмы обработки массивов.

type

tStack=tPseudoArray;

procedure Pop(var stack:tStack; var x:T);

begin with stack do begin top:=top-1; x:=Content[top] end; end;

procedure Push(var stack:tStack;x:T);

begin with stack do begin Content[top]+1; top:=top+1; end; end;

{при неосмотрительном использовании, выполнение операторов чревато }

{выходом за границы массива [1..MaxLen]}

{но ситуация не совсем симметрична, у пользователя есть функция проверки пустоты стека, но нет функции проверки переполнения стека }

```
function Empty(Stack:tStack):boolean;
begin Empty:=Stack.top=1 end
procedure Create(var Stack:tStack); begin Stack.top:=1 end;
procedure Destroy(Stack:tStack) ); begin Stack.top:=1 end;
```

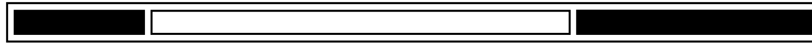
Одинаковая реализация разных операций, конечно, настораживает. Create призвана порождать *функцию* (с пустой областью определения), Destroy - уничтожать *функцию* (с любой областью определения), наша релизация лишь опустошает *область* определения функции. Причина ясна - мы никак не моделируем понятие состояния (см. далее) Пока оставим нюансы - так или иначе, главные стековые операции push и pop работают правильно.

Обратимся к реализации очередей. Определим "псевдодинамические" массивы с двумя концами.

```
tPseudoArray2=
record      content:tArray; {содержимое/компоненты массива}
      start, finish:tIndex;       {начало+1 и конец-1 массива -}
                        {начало правой кучи и конец левой кучи}
end;
tQueue=tPseudoArray2;
```

Реализация операций как будто очевидна - класть значения в конец, а брать - из начала массива. Формально верная, такая реализация порождает частный случай *проблемы динамического распределения памяти* (общую формулировку см. ниже): Вводя в конец (занимая одну, "правую" часть кучи) и выводя из начала массива значения компонент (опустошая *другую,* "левую" часть кучи), мы весьма скоро можем оказаться в ситуации, когда свободной памяти много, а класть компоненты некуда!

Правда, в этом частном случае ее нетрудно решить, объединяя две части кучи, мысленно рассматривая массив как кольцо.



procedure ***Put(var Queue:tQueue; x:T); { Поставить В Очередь }***

begin

with Queue do

begin content[finish]:=x;

if finish=nMax then finish:=1 else inc(finish) {≈ finish:=finish+1 (mod nMax)}}

{интересно - см. понятие модульной арифметики в курсе дискретной математики}

end; end;

procedure ***Get(r,x); { Вывести Из Очереди }***

begin

with r do

begin x:= content[start];

if start=1 then start:=nMax else dec(start) {≈ start:=start-1 (mod nMax)}}

end; end;

function ***Empty(r):***boolean;

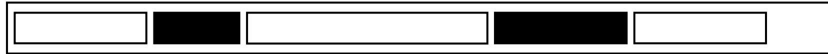
begin Empty:= (start=finish) or ((start=nMax) and (finish=1)) {start=finish (mod nMax)} end;

Замечание. Снова более эффективная, но не *защищенная* реализация - пользователь процедур должен сам следить за переполнением очереди.

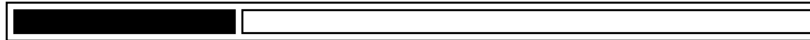
Проблема распределения памяти. Списочные структуры.

Проблема распределения памяти - проблема ее фрагментированности ("проблема кучек"):

- необходимо сохранить некоторую последовательность f,
- *но* нет ни одной кучки (сплошного незанятого участка памяти), достаточной для хранения всех элементов последовательности f в естественном порядке,
- *хотя* совокупного объема свободной памяти для этого достаточно.



Черные области - занятые участки памяти (область определения массива памяти как некоторой последовательности), белые - незанятые (область неопределенности), внешняя рамка - некоторый интервал [1..N] (область имен/указателей/индексов на участки память).



Очевидный вариант решения проблемы - *дефрагментация* - копирование полезной информации в одну, а куч свободной памяти - в другую часть памяти - действительно, применяется, но очевидно, крайне трудоемок.

Надо придумать способ хранения компонент последовательности $f:N \rightarrow T$, не зависящий от порядка расположения компонент (в массиве/памяти). Цель - уметь класть очередную компоненту на произвольное (первое попавшееся) свободное место. Необходимое и одновременно изящное решение - хранить f в виде функции $F: N \rightarrow N \times T$ с произвольной (т.е. "дырявой" и не упорядоченной) областью определения $Dom(F)=\{n_1, n_2, ..., n_k\}$, такой, что

$$(*)\ F(n_1)=\langle n_2,f(1)\rangle,\ F(n_2)=\langle n_3,f(2)\rangle,..,\ F(n_i)=\langle n_{i+1},f(i)\rangle,\ldots\ F(n_k)=\langle n_{k+1},f(k)\rangle$$

Такой способ хранения последовательностей называется списковой организацией памяти, или просто ***списком.*** По определению, список F хранит значения f и индекс (указатель,"имя") следующего ее значения. Указатель n_1 называют *головой* списка, указатель n_{k+1}, *не* принадлежащий Dom(F) - *признаком конца списка.* Обычно, в качестве признака выделяют специальный

"пустой" указатель **0** (например, число 0 или -1), единственный смысл которого - *ни на что не указывать* (по определению, **0** $\notin$Dom(F) для всех списков F).

Основные операции над списками - перечисление, вставка и удаление компонент - *никак* не используют арифметические операции на Dom(F), т.е. тот факт, Dom(F)$\subset$N, а лишь то, что они, в качестве имен, указывают (ссылаются) на значения. Это и делает возможным реализацию списков ссылочным типом.

```
type
    tComponent=record value:T;next:p:pComponent end;
    pComponent=^tComponent;
    pList=pComponent;
{список задается ссылкой на голову - первую компоненту списка}
{или значением nil,  если такового не существует, т.е. список пуст}

{перечисление компонент списка}
procedure Enumeration(List:pList);
var pt:pComponent;
{если pt≠nil,  pt^.value=компонента хранимой последовательности fi}
begin
    pt:=List; {≈ i:=1}
    while not (pt<>nil) {≈ i<=длины f}  do
    begin {Обработка fi = pt^.value } pt:=pt^.next {≈ i:=i+1} end;
end;

{порождение списка (здесь: из компонент файла f: file of T)}
procedure Generation(var List:pList; var f: FileOfT);
var pt:pComponent;
{если pt≠nil,  pt^.value=компонента хранимой последовательности fi}
begin
    reset(f); List:=nil; {≈ i:=0}
```

```
        while not eof(f) {≈ i<=длины f}  do
        begin new(pt); read(f, pt^.value); { pt^.value:= fi }
              pt^.next:=List; List:=pt
        end;
  end;
```

Здесь список хранит компоненты исходной последовательности в обратном порядке, что не всегда приемлимо и удобно. Мы обязаны хранить ссылку на начало (первую компоненту) списка, но мы можем хранить ссылки и на другие его компоненты. Такие компоненты можно назвать *активными.*

Реализацию операций вставки и исключения из списка - см. "Задачи текстовой обработки"

Реализация стеков и очередей списками.

Применим общее решение проблемы распределения памяти в виде списков к реализации абстрактных линейных типов (АТД).

Реализация стека вполне тривиальна.

```
{АТД стек}
type
tComponent=record value:T;next:p:pComponent end;
pComponent=^tComponent;
pStack=pComponent; {ссылка на голову - первую или "верхнюю" компоненту списка}

procedure create(var Stack: pStack); begin   Stack:=nil  end;
procedure destroy(var Stack: pStack);
var p:pComponent;
begin
{сборка мусора }
while Stack<>nil do
```

```
begin
        p:=Stack; stack:=stack^.next; dispose(p);
end;

end;
procedure push (var Stack: pStack; x:T);
var p:pComponent;
begin
        new(p);  p^.value:=x;  p^.next:=Stack;  Stack:=p
end;

{ стек предполагается непустым }
procedure pop( var Stack: pStack; var x:T);
var p:pComponent;
begin
p:=Stack;  x:=Stack^.value; Stack:=Stack^.next; dispose(p);  {сборка мусора}
end;

function empty(Stack: pStack): boolean; begin empty:= Stack=nil end;
```

Реализация очередей следует затронутой ранее идее "закольцованной" последовательности, представленной кольцевым списком. Так называют список, последняя компонента которого ссылается на первую.

```
{АТД очередь}

type
pQueue= pComponent; {указатель на начало кольцевого списка }
{создание пустой очереди; sic - ей соответствует не пустой список}
procedure Create(var Queue: pQueue);
begin
```

```
new(Queue); {нулевой элемент - фиктивный}
Queue ^.next:= Queue;
end;end;

procedure Put(var Queue: pQueue; x: T); { Поставить В Очередь }
var p:pComponent;
begin
{вставляем новое значение x ДО нулевой}
Queue ^.value:=x;
{вставляем новую компоненту p после нулевой}
new(p); p^.next:= Queue ^.next; Queue ^.next:=p;
{делаем ее нулевой}
Queue:= p;
end;



function Empty(Queue:pQueue):boolean;
begin     empty:= Queue = Queue ^.next end;

procedure Get(Queue: pQueue; var x: T);{ Вывести Из Очереди }
{предполагается, что очередь не пуста; аналогична pop, но надо не забыть про фантом}
var первый:pComponent;
begin
первый:= Queue ^.next;
x:=первый^.value;
Queue ^.next:=  первый^.next;
dispose(первый); {сборка мусора}
end; end;
```

```
procedure Destroy(Queue:pQueue);
var первый,p:pComponent;
begin
{сборка мусора }
первый:= Queue ^.next;
while первый<> Queue do
begin
	p:=первый; первый:= первый ^.next; dispose(p);
end;
dispose(Queue)
end;
```

§10. Алгоритмы полного перебора.

Пусть T - множество значений некоторого порядкового типа, T={Первый<Второй<…Последний}, succ - соответствующая функция следования, а Seq(L)=Seq(T,L)=[1..L]→T - множество всех последовательностей длины L.

Лексикографический, или словарный порядок на Seq(L) определяется следующим образом. Пусть a,b $\in$ Seq(n), a$\neq$b и N=N(a,b)=min {n: a(n) $\neq$b(n)}- наименьший номер i, на котором они различаются. Тогда a считается меньшей b, если у a на N-ом месте стоит меньшая (в смысле порядка на T) компонента, чем у b: a<b $\approx$def a(N)$<_T$ b(N).

Пример. T=кириллица (русский алфавит), последовательности символов - слова. Тогда 'шабаш'< 'шалаш' (в Seq(T,5)).

Алгоритм определения следующего члена последовательности.

Следующая(a)=b $\approx$ найдется число N такое, что для всех i, i $\in$ [1..N], b(i)=a(i), b(N)=succ(a(N)) и b(j)=Первый, для всех j, j $\in$ [N+1..L]. Причем, нетрудно заметить (и доказать), что N в этом случае определяется однозначно - N=max{i:a(i) $\neq$Последний}.

Идея вычисления функции Следующая:

а) Вынимай *из конца* последовательности a все последние (в T) значения, пока не найдешь меньшего значения c. Если такого значения c нет, то a - это последняя последовательность.

б) Положи *в конец* a значение succ(c)

в) Доложи *в конец* последовательности необходимое число первых значений.

Пример. Построим следующее за 00011 слово (в Seq({0<1},5) - после шага а) получаем 00, после б) 001, после в) - 00100.

Обработка последовательностей "с одного конца", как мы помним, реализуется в терминах стеков (см. "Абстрактные линейные типы").

```
{exists,a:=∃ b (b=Следующая(a)), Следующая(a)}
{ясно, exists=∃ j∈ [1..LenA] (a(j)<>Последний}
{pa - ссылка на содержимое стека, содержащего последовательность a}
{LenA - константа, длина последовательности}
procedure Следующая(pa:pSeq; exists:boolean);
var
i:integer; {число вынутых компонент}
x:T; {значение последней компоненты}
begin
i:=0; exists:=false;
{a} while (i<=LenA) and not found do
begin pop(pa,c); i:=i+1; if c<>Последний then exists:=true end;
if exists
then   begin {b} push(pa,succ(c));
              {c} while i<>0 do begin push(pa,Первый);i:=i-1 end
       end;
```

Любую задачу с конечным числом вариантов решений можно представить как поиск пути в некотором графе из некоторого множества инициальных (исходных, «данных») в некоторое множество финальных (конечных, желаемых). Перечисление последовательностей дает далеко не

эффективный, но *универсальный* алгоритм решения таких конечных задач полным перебором всевозможных путей решения.

Задача о раскраске графа. Найти ("распечатать") все правильные раскраски произвольного графа с фиксированным числом вершин n (если таковые найдутся) в m цветов. Раскраска правильная, если никакие две соседние (т.е. связанные некоторым ребром) вершины не окрашены в один цвет.

Предварительный анализ

found=∃ r ∈ tРаскраска (правильная(r))

где Правильная(r)= ∀ i,j∈ Вершины (Соседи(i,j)→Цвет(r,i)≠ Цвет(r,j))

Следующий алгоритм полного перебора раскрасок - тривиальное кратное обобщение задачи поиска (см. "Поиск" и "Вычисление свойств")

```
function Правильная(r:tРаскраска):boolean;
begin
b:=true;
i:=ПерваяВершина;
while (i<=pred(ПоследняяВершина)) and not b do
begin j:=succ(i);
    while (j<=ПоследняяВершина) and not b do
    if Соседи(i,j) and (Цвет(r,i)=Цвет(r,j))
then b:=false else j:=succ(j);
i:=succ(i)
end;

procedure ПолныйПеребор(graph:tGraph; var found:boolean);
var r: tРаскраска; exists:boolean; {}
```

```
begin
r:=ПерваяРаскраска; found:=false; exists:=true;
while exists (= not КончилисьРаскраски} do
begin if Правильная(r) then begin found:=true; Печать(r) end
        Следующая(r,exists) {r:=Следующая(r)}
end;
end;
```

Дальнейший анализ.

Тип tВершины должен быть порядковым - можно, например пронумеровать все вершины - tВершины=1..n;

Тип tGraph хорошо бы реализовать так, чтобы было легко вычислять функцию Соседи: tВершины × tВершины → Boolean. Хороший вариант - задать граф матрицей инциндентности (т.е. табличным определением функции Соседи)

tGraph=array[tВершины, tВершины] of Boolean;

Тип tРаскраска хорошо бы реализовать так, чтобы было легко вычислять цвет каждой вершины r: tВершины→tЦвет (реализация типа tЦвет произвольна, лишь бы было определено равенство, пусть tЦвет=1..m).

Хороший вариант tРаскраска=array[tВершины] of tЦвет= array[1..n] of tЦвет, но тип array - *не порядковый*, что требуется нашим алгоритмом. Но - теперь мы умеем организовать перебор последовательностей с помощью стековых операций.

Вывод - реализовать раскраски как стек (который, в свою очередь, в данном случае лучше реализовать массивом).

Опустив детали реализации, подведем некоторые предварительные итоги решения. Алгоритм прост, но малоэффективен. Перебор всех m^n раскрасок - функций/последовательностей [1..n]→[1..m] - дело долгое.

§ 11. Перебор с возвратом.

Как сократить область перебора в переборных алгоритмах? Вернемся к решению задачи о раскрасках карты из § 12 (здесь - повторить постановку и вкратце идею и «дефект» решения алгоритмом полного перебора).

Идея сокращения проста, но изящна - рассматривать *неполные* раскраски - динамические последовательности, т.е. функции $[1..k] \rightarrow [1..m]$, $k \leq n$. Основание - если (неполная) раскраска r уже не правильна, нет никакого смысла терять время на перебор раскрасок большей длины.

Для этого доопределим ***лексикографический порядок*** на последовательностях Seq произвольной длины, $Seq = \cup \{Seq(L): L \in N\}$. Пусть снова $a,b \in Seq$ - теперь, вообще говоря, разной длины LenA и LenB, $a \neq b$ и $N=N(a,b)=\min \{n: a(n) \neq b(n)\}$- наименьший номер i, на котором они различаются. Тогда a<b, если

a) Либо $N \in Dom(a)$ ($N \leq LenA$), $N \in Dom(b)$ ($N \leq LenB$) и $a(N) <_T b(N)$.
b) Либо $N \in Dom(a)$ ($N \leq LenA$) и $N \notin Dom(b)$ ($N > LenB$), т.е. a - начальный отрезок последовательности b

(можно свести все к "старому" случаю a), если мысленно доопределить последовательность меньшей длины добавленным в T "пустым" значением $\varnothing$, сделав его минимальным $\varnothing$ =Нулевой=pred(Первый))

Пример. 'шабаш'<'шабашка', 'шабаш'<'шалаш'

Алгоритм определения следующей последовательности - полностью повторяет прежний, *за исключением* пункта в) (теперь нам не зачем доопределять следующее значение до фиксированной длины). Операцию перехода к следующему значению в таком порядке принято называть *возвратом.*

```
procedure ПереборСВозвратом(graph:tGraph; var found:boolean);
var r: tРаскраска; exists:boolean;
begin
r:=ПерваяРаскраска; {заметь - теперь это пустая последовательность!}
found:=false; exists:=true;
while exists {= not КончилисьРаскраски длины менее n} do
begin if Правильная(r)
        then if Полная(r) {т.е. длины n}
                then begin found:=true; Печать(r) end
                else {дополнить - добавить первое значение в T}
                        Push(r, Первый)
        else {не правильная} Возврат(r, exists) {=Следующая(r,exists)}
end;
end;
```

Замечание. Экзаменатор вправе потребовать довести реализацию до конца!

§12. Нелинейные типы данных.

Существует много определений одних и тех же (с точки зрения математики) содержательных (интуитивных) понятий. Так, под графами мы содержательно подразумеваем визуальные (зрительное) понятие - диаграммы, состоящие из вершин и соединяющих их ребер - отрезков (неориентированный граф) или стрелок (ориентированный граф).

С формальной стороны, (ориентированный/неориентированный) ***граф*** можно отождествлять с множеством его ребер Arrows, а последнее, в свою очередь с некоторым (произвольным/симметричным) бинарным отношением, т.е. подмножеством декартового квадрата (множества всех пар) Nodes×Nodes:

$\forall$ a,b$\in$ Nodes

(<a,b>$\in$ Arrows $\approx$ в графе есть стрелка, ведущая из вершины a в вершину b)

Так определенный граф мы, скорее всего, реализуем т.н. ***функцией инциндентности*** (соседства) - предикатом Nodes×Nodes→Boolean, а последний - булевским массивом

```
type
tArrows=array[tNodes,tNodes] of boolean;
{при реализации симетричного отношения желательны треугольные массивы, в Паскале отсутствующие}
```

С другой стороны, граф можно отождествлять с ***функцией перехода*** из вершины по данному ребру Go$\in$Arrows×Nodes→Nodes, для некоторого множества ребер Arrows и множества вершин Nodes:

$\forall$ r$\in$ Arrows $\forall$ a,b$\in$ Nodes (Go(r,a)=b $\approx$
ребро r ведет из вершины a в вершину b)

Так определенные диаграммы мы будем называть ***автоматами*** (без выхода). При этом вершины из Nodes обычно понимаются как маркировка/именование состояний некоторого объекта, а стрелки - маркировкой некоторых элементарных преобразований состояний.

Понятно различие двух определений. В первом мы (явно ли нет) именуем/указываем/помечаем лишь множество вершин, во втором - и множество ребер. Для программиста важно осознавать и различие определений (они ведут к разным типам-реализациям), так и возможность их отождествления (преобразования типов с сохранением семантики).

Работа (функционирование) автомата описывается индуктивно определяемой функцией $Go^* \in Arrow^* \times Nodes \rightarrow Nodes$. При этом конечные последовательности из Arrow* понимаются как пути в диаграмме или же - трассы преобразований состояний (см. и сравни "Поиск").

```
type
tArrows=1..nMaxArrows;
tNodes=1..nMaxNodes;
{реализация автомата массивами}
tAutomaton=array[tArrows,tNodes] of tNodes;
{реализация автомата ссылками}
pAutomaton=pNode; {ссылка на начальное состояние автомата - см. a),b) ниже}
pNode=^tNode;
tNode=record
Content:T;{содержимое вершины/ определение значения состояния}
        Arrows:array[tArrows] of pNode;
{последовательность исходящих стрелок, реализованная массивом}
{динамический вариант - линейный список}
```

end;

Здесь нас интересуют графы/автоматы специального вида - бинарные деревья (общую теорию графов и автоматов см. курс дискретной математики). Автомат Go - ***дерево***, если существует ***корневая*** вершина (начальное состояние автомата) $\varnothing$,

a) в которую не ведет ни одно ребро, $\forall\ a\in$ Nodes ($Go(r,a)=\varnothing \rightarrow a=\varnothing$),
b) но из которой существует путь в любую вершину,
   $\forall\ a\in$ Nodes $\exists\ r\in$ Arrows ($Go(r,\varnothing)=a$),

причем этот путь - единственный, $\forall\ r,r'\in$ Arrows ($Go(r,\varnothing)=Go(r',\varnothing) \rightarrow r=r'$).

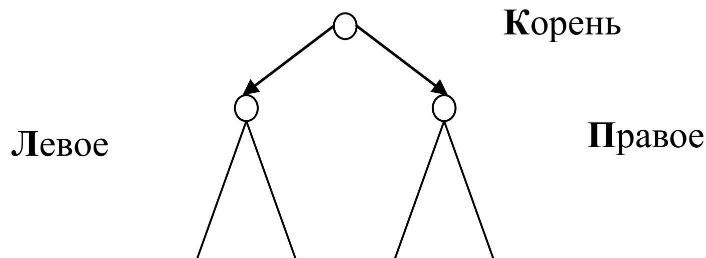
Go - ***бинарное дерево***, если, к тому же, множество стрелок состоит из лишь из двух стрелок, далее обозначаемых как левая (left) и правая (right), Arrows={left,right}.

Другие возможные обозначения: Arrows={0,1} или Arrows={true,false}. При желании и возможности дайте также графовое определение понятие дерева - в частности, *рекурсивное* определение, соответствующее диаграмме ниже.

```
{реализация бинарного дерева ссылками}
pTree=pNode; {дерево задается ссылкой на корень}
pNode=^tNode;
tNode=record
Content:T;{содержимое вершины}
        left,right: pNode;
end;
```

Задача обхода дерева заключается в переборе его вершин в некотором порядке, т.е. построении некоторой функции следования (счета) на Arrows*×Nodes.

Самый простой, но исключительно важный класс таких порядков связан с *лексикографическим* порядком на множестве "слов"/путей Arrows* (см. "Перечисление последовательностей"), связанных со всеми возможностями определения такого порядка на "алфавите" - множестве Arrows.



(По старой программистской традиции мы рисуем деревья "вверх тормашками"). Так, при КЛП-обходе для любого поддерева исходного дерева корень обрабатывается (идет в перечислении раньше), чем все нижние вершины, причем все вершины его левого поддерева обрабатываются раньше, чем все вершины правого. Аналогично определяются ЛПК, ЛКП, ПЛК, ПКЛ и КПЛ обходы.

Чуть точнее: добавим во множество Arrows пустой элемент ∅ (соответствующей стрелке из каждой вершины в себя) и рассмотрим всевозможные порядки на множестве Arrows={∅,left,right}. Любой такой порядок продолжается на множество путей Arrows* - конечных последовательностей $<r_1,..,r_L>$, рассматриваемое теперь как множество функций r с *бесконечной областью определения N* таких, что $r(i)= r_i$, для $i\in [1..L]$ и $r(j)=\varnothing$, для $j>L$. Положим теперь $r1<r2 \approx r1\neq r2$ и $r1(i_0)<r2(i_0)$ (в смысле выбранного порядка на Arrows), $i_0=\min \{i: r1(i)\neq r2(i)\}$. Например, при сравнении 3-х путей 'Left', 'LeftLeft' и 'LeftRight' {∅<left<right} влечет 'L'<'LL'<'LR', {left<right<∅} - 'LL'<'LR'<'L' и {left<∅<right} - 'LL'<'L'<'LR'.

Определение обхода в терминах лексикографического порядка дает следующую идею алгоритма обхода заданием приоритета выбора вершин:

Шаг 0. Начни с первого, в данном порядке, пути

В одних порядках это - пустой путь к корню дерева, в других его необходимо предварительно построить!

Шаг i+1

(если нет никаких направлений - обход окончен, иначе)

(если можно идти в направлении D_1,) иди в направлении D_1,

(если нет, но можно идти в направлении D_2,) иди в направлении D_2,

(если нет, но можно идти в направлении D_3,) иди в направлении D_3,

Здесь D_1,D_2,D_3 принадлежат алфавиту {up ("вверх"), left ("налево"), right (направо)}, нумерация задается выбранным на нем порядком, причем ход наверх соответствует значению $\varnothing$.

Реализация спуска налево и направо от текущей вершины очевидна. Главная трудность - в реализации подъема: в деревьях все пути ведут "вниз"!

Идея

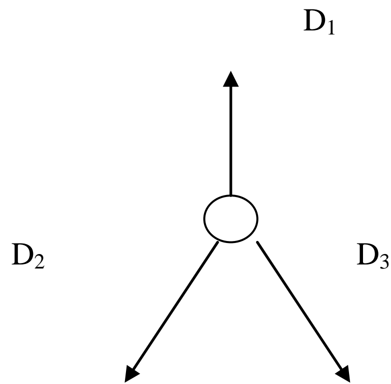
- нужно запоминать ссылки на корни деревьев, которые еще предстоит обработать - самое простое - *класть* их в конец некоторой (динамической!) последовательности.
- для реализации подъема нужно *вытащить* ссылку на вершину из хранилища; в зависимости от вида обхода, это может означать как "вытаскивание" как с *того же*, так и *другого* конца последовательности.

Сказанное определяет тип последовательности-хранилища: это либо ***стек***, реализующий "вытаскивание" с того же конца, либо ***очередь,*** реализующая "вытаскивание" с противоположного конца. (См. *"Абстрактные линейные типы").*

{пример ***обхода "в глубину"*** ≈ ∅ < {left,right} }

{≈ более короткие ветки (слова, пути) идут в перечислении раньше более длинных}

```
procedure КЛП(root:pTree);
var
pt:pNode;{ссылка на текущую вершину}
stack:pStack; { реализация - см. "Абстрактные линейные типы"}
begin
Create(stack);
if root<>nil then push(stack,root);
while not empty(stack) do
begin
    pop(stack,pt); {подъем наверх}
    with pt^ do
    begin
    { какая-то обработка содержимого текущей вершины pt^.value}
    if left<>nil {можешь идти налево?}
    then {тогда иди, но сохрани правую ссылку}
    begin pt:=left; if right<>nil then push(stack,right) end
    else {налево нельзя}
    if pt^.right<>nil {- можно ли направо?}
    then pt:= pt^.right
    end;
Destroy(Stack)
end;
```



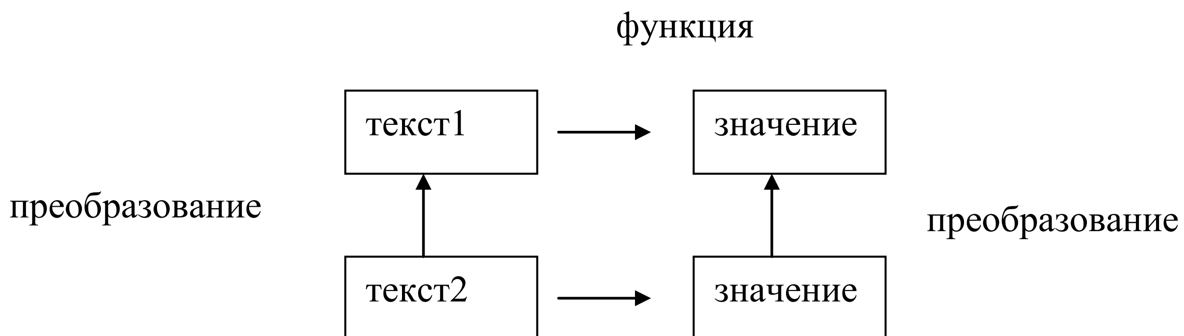
{пример ***обхода в ширину*** - ветви одинаковой длины ("буквы" left, right) соседствуют в перечислении}

```
procedure (root:pTree);.
```

```
var
Queue:pQueue; { реализация - см. "Абстрактные линейные типы"}
pt:pNode;{ссылка на текущую вершину}
begin
Create(Queue); {создать очередь}
if root<>nil then Put(Queue,root); {Поставить В Очередь }
while not Empty(Queue) {очередь не пуста} do
begin
        Get(Queue, pt); {Вывести Из Очереди}
        {обработать содержимое pt^.value вершины pt}
        if pt!.left<>nil then Put(Queue, pt!.left);
        if pt!.right<>nil then Put(Queue, pt!.right);
end;
Destroy(Queue)
end;
```

Замечание. Деревья просто определяются рекурсивно, а потому – для проявления эрудиции – нетрудно дополнить ответ рекурсивными вариантами обхода «в глубину». Но – не заменять ими итеративный вариант!

Последние 3 вопроса относятся к алгоритмам символьной обработки. Важность задач обработки символьных последовательностей (текстов) связана с *универсальностью* текстов как имен (обозначений). Значение *любого* типа как-то выражается, определяется текстом его записи (в некоторой системе обозначений).



В этой связи круг естественно возникающих при обработке текстов задач можно разделить на 3 части.

1) Вычисление *функции значения/именования,* т.е. нахождение значения по его обозначению и обратная задача.
2) *Формальная обработка* («редактирование») текстов как собственно последовательностей, никак не связанная с ролью текстов как обозначений.
3) *Формальные вычисления* - порождение текста результата некоторого преобразования значений по тексту ее аргументов

§ 13. Алгоритмы обработки выражений.

Определим ***арифметическое выражение*** (в полноскобочной инфиксной записи) как последовательность символов, являющуюся

1) либо *термом*, т.е.

a) либо именем константы вида $b_1 \ldots b_n.c_1 \ldots c_m$, где

$$(n>0) \& (m>0) \& \ \forall\, i\in[1,n](b_i\in [\text{'0'},\text{'9'}]) \& \forall\, j\in[1,m](c_j\in [\text{'0'},\text{'9'}])$$

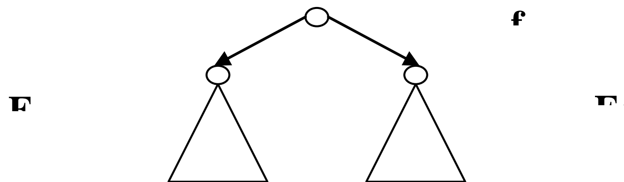
b) либо именем переменной вида $d_1 \ldots d_k$, где

$$(k>0) \& (d_1\in [\text{'a'},\text{'z'}]) \& \forall\, i\in[2,k](d_i\in [\text{'0'},\text{'9'}]\cup[\text{'a'},\text{'z'}])$$

2) либо строкой вида $(E_1)f(E_2)$, где E_1,E_2 - выражения, а f - один из знаков +,-,*,/

Аналогично определяются выражения в префиксной $f(E_1)(E_2)$, и постфиксной $(E_1)(E_2)f$ формах.

Определим также ***дерево выражения*** E как дерево над базовым типом



string, состоящее из единственной вершины, содержащей E, если E - терм или же дерево вида

если E - выражение вида $(E_1)f(E_2)$.

Аналогично определяются деревья выражений в префиксной и постфиксной формах. Ясно, что вид дерева не зависит от синтаксиса/структуры сложного выражения. В чем и заключается их прелесть. Минус – больший расход памяти, по сравнению в линейной, строковой форме.

Как перейти от представления выражения деревом к линейной - инфиксной, префиксной и постфиксной форме? Ответ ищи в разных порядках обхода дерева в "Нелинейные типы данных".

Типом выражения E (и, соответственно, вершины, его содержащей) будем называть символ-метку 't', если E - терм, и символ 'f', если E - выражение, содержащее знак операции.

Задача вычисления дерева выражения. Найти результат выражения при заданных значениях переменных, если выражение представлено деревом.

Строго говоря, для решения задачи мы должны также зафиксировать *семантику* выражений - что же они на самом деле обозначают? Ясно, что по умолчанию подразумевается традиционная арифметика вещественных чисел.

Для того, чтобы кратко и ясно представить основную идею алгоритма как сведение вычисления сложного выражения к вычислению *элементарных* выражений или *атомов* (содержащих, по определению, единственный знак операции), будем предполагать, что значения термов (констант и переменных) уже найдены (как это сделать - см. "Задачи текстовой обработки,3") и сохранены в самом дереве.

```
type
pNode=^tNode;
tNode=      record
            Name:string;
            Val:integer;
            left,right:pNode;
end;
```

Очевидно, это не очень экономное решение в смысле расхода памяти - для двух термов с одинаковым значением мы сохраняем значение дважды, а остальные (содержащие знаки операций) вершины вообще изначально не имеют значения. Обычно значения термов и промежуточные результаты сохраняют во внешних по отношению к дереву таблицах.

```
{вычисление элементарного выражения}
function AtomVal(root:pExpr):T;
begin
with root^ do
case Name of
```

```
        '+': AtomVal:=left^.Val+right^.Val;
        '-': AtomVal:=left^.Val-right^.Val;
        '*': AtomVal:=left^.Val*right^.Val;
        '/': AtomVal:=left^.Val/right^.Val;
        {для простоты, опускаем обработку исключения - деления на 0}
end; end;

{выяснение типа вершины}
function NodeType(pt:pNode):char;
begin
with pt^ do
if (right=nil) and (left=nil)
then NodeType='t'
else NodeType='f'
end;

{поиск атома, лежащего ниже данной вершины}
{идея: вычисление ∃-свойства found:=∃ pt∈pNode (pt≠nil & pt^.left≠nil  &
pt^.right≠nil) }
{см."Вычисление свойств"}
{предусловие: вершина pt содержит знак операции, NodeType(pt)='f'}
procedure FindAtom(var pt:pNode; var stack: pStack);
var found:Boolean;
begin
with pt^ do
begin
found:=false;
while not found do
begin
if NodeType(left) ='f'
```

```
then begin push(pt,stack); pt:=left end
else if NodeType(right)='f'
    then begin push(pt,stack); pt:=right end
    else
    { это терм ≈ NodeType(pt)='f' & NodeType(left) ='t' & NodeType(right)='t'}
    found:=true
end;
end;
{предусловие - дерево не вырождено, все термы уже означены}
function ExpVal(root:pExpr):T;
var
stack:pStack;
{вспомогательный стек ссылок pNode на еще не вычисленные знаки операций}
{реализацию см.  "Абстрактные линейные типы данных"}
found:Boolean; {есть еще атомы?}
RootType,LeftType,RightType:char; {{'t','f')}
begin
Create(stack);  UpperType:=NodeType1(root);
if UpperType='f' then push(stack,root);
while not empty(stack) {= есть невычисленные операции} do
begin
{следующий поиск начинаем с вершины, ближайшей к  вычисленному атому}
pop(stack,pt); FindAtom(pt,stack);
{Вычисли значение атомарного выражения}
pt^.val:= AtomValue(pt);
{Сократи дерево}
destroy(pt^.right); pt^.right:=nil;
```

```
destroy(pt^.left); pt^.left:=nil;
end;
{Дерево состоит из одной вершины}
ExpVal:=root^.val;
end;
```

Замечание. Побочным (и не очень желательным) эффектом простоты нашего алгоритма является уничтожение самого дерева. В реальности, мы можем не уничтожать вычисленные термы, а лишь помечать их как уже вычисленные.

Задача. Синтаксический анализ выражения, представленного деревом. Проверить, является ли произвольное бинарное символьное дерево деревом выражения.

Задача синтаксического анализа очевидно распадается на анализ 1) термов и 2) сложных выражений. В данном случае, анализ термов прост и напрямую сводиться к задаче поиска (см. "Поиск"). *Центральная идея:* свести задачу анализа сложного выражения к задаче вычисления.

1) *Анализ термов.*

```
type tSetOfChar=set of char;

{Поиск позиции "исключения" - первого символа в подстроке s[start,finish], не принадлежащего множеству alphabet }
procedure FindException
(s:string; start,finish:Cardinal; alphabet:tSetOfChar; position:Cardinal; found:boolean);
```

```
    {идея: проверка свойства found=∃ position∈ [start,end] (s[position] ∉ alphabet)}
    {см."Вычисление свойств" и "Поиск"}
    begin
        found:=false; position:=start;
        while (position<=finish) and (not found) do
        if s[position] in alphabet then inc(position) else found:=true;
    end; { function FindException }

    {проверка имени константы}
    function ConstantName(s:string):boolean;
    var
        position,     {позиция "исключения" - см. procedure FindException }
        len:Cardinal;        {длина выражения s}
        found:boolean;      {"исключение" найдено}
    begin
        len:=Length(s); ConstantName:=false;
        FindException(s,1,len,['0'..'9'],position,found);
        if (position=1) or (position=len) or (not found)
        then {нет обязательной внутренней не-цифры} exit; {завершаем}
        if s[position]<>'.'
        then {эта не-цифра - не точка}  exit; {завершаем}
        {есть ли не-цифры после точки?}
        FindException(s,position+1,len,['0'..'9'],position,found);
        ConstantName:=not found
    end;

    {проверка имени переменной - еще проще}
    function VariableName(s:string):boolean;
    var
```

```
        position,      {позиция "исключения" - см. procedure FindException }
        len:Cardinal;        {длина выражения s}
    found:boolean;     {"исключение" найдено}
    begin
        len:=Length(s); VariableName:=false;
        if len=0 then exit;
        if not (s[1] in ['a'..'z'])     then exit;
        FindException(s,2,len,['0'..'9']+['a'..'z'],position,found);
        VariableName:=not found
    end;
```

2) Анализ (сложных) выражений.

Определим понятие расширенного (или просто р-) выражения как строки вида $(E_1)F(E_2)$, где E_1,E_2 - произвольные, в том числе *пустые* строки, а F - произвольная *непустая* строка. Мотивация - *любому* невырожденному дереву однозначно сопоставляется некоторое р-выражение.

Расширим также понятие типа р-выражения за счет добавления "неопределенного" значения '∅' : если р-выражение не имеет типа 'f' или 't', то оно имеет тип '∅'. Формальным вычислением р-выражения назовем операцию Reduce (сократить, англ):

$$\text{Reduce}((E_1)F(E_2)) = \begin{cases} \text{терм 'x'}, & \text{если Тип}(E_1)=\text{Тип}(E_2)=\text{'t' и Тип}(F)=\text{'f'} \\ \text{'}\varnothing\text{'}, & \text{в противном случае} \end{cases}$$

(смысл прост - результатом вычисления правильного выражения является терм, а результат вычисления неправильного выражения не определен)

Идея алгоритма: Выражение правильно ≈ тип результата формального вычисления = терм.

```
{расширенный тип вершины}
function ExtentedNodeType(pt:pNode):char;
var len:Cardinal; {длина строки}
begin
{теперь не уверены, что "листья" дерева - это термы!}
ExtendedNodeType:='∅';
if pt=nil  { тип пустого слова неопределен}
then exit; {= завершить определение значения функции }
with pt^ do
begin
len:=Length(name);
if len=0 then exit;
if  (Len=1) and (name[1] in ['+','-','*','/']) and (left<>nil) and (right<>nil)
        then begin ExtendedNodeType:='f'; exit end;
if (left=nil) and (right=nil) and
   (ConstantName(pt^.name) or VariableName(pt^.name)
        then ExtentedNodeType:='t'
end; {with}
end; { function ExtentedNodeType }


{теперь не уверены, что найдутся правильные атомы!}
{но, в таком случае, обязательно найдется не правильный }
{см. определение ниже в теле процедуры}
procedure FindExtendedAtom(var pt:pNode; var stack: pStack);
var found:Boolean;
begin
with pt^ do
begin
```

```
found:=false;
while not found do
begin
        UpperType=NodeType(pt);
        LeftType= NodeType(left);
        RightType=NodeType(right);
        if (UpperType<>'f') or (LeftType='∅') or (RightType='∅')
        then
        {дальше искать нечего, формально считаем - найден "не
правильный" атом}
        found:=true
        else if LeftType='f'
        then begin push(pt,stack); pt:=pt^.left end
        else if RightType='f'
                then begin push(pt,stack); pt:=pt^.right end
                else   {UpperType='f' & LeftType='t' & RightType='t'}
                        { ≈ найден правильный атом }
                        found:=true
end; {while}
end; {with }
end; { procedure FindAtom }

{формальное вычисление расширенного элементарного выражения}
function ExtendedAtomVal(root:pExpr):string;
begin
with root^ do
if      (ExtendedNodeType(root)='f') and
(ExtendedNodeType(left)='t') and
(ExtendedNodeType(right)='t') {корректный атом}
then  ExtendedAtomVal:='x' {сгодиться любой терм!}
```

```
else  ExtendedAtomVal:='∅';
end;

function ExpressionCorrect(root:pExpr):boolean;
begin
result:=true;
create(stack);
RootType:=ExtendedNodeType(root);
case RootType of
        '∅': result:=false;
        'f': push(stack,root);
end;
while (not empty(stack)) and result do
begin
pop(stack,pt);
FindExtendedAtom(pt,stack);
{Формальное "вычисление"}
pt^.Name:=ExtendedAtomValue(pt);
if NodeType(pt)='∅'
        {если результат элементарного вычисления неопределен}
then Result:=false {то результат всего вычисления - и подавно}
else
begin {Сократи дерево}
        destroy(pt^.right); pt^.right:=nil;
        destroy(pt^.left); pt^.left:=nil;
end; {if}
end; {while}
Result:=(NodeType(root)='t')
end;
```

§ 14. Алгоритмы текстовой обработки.

См. преамбулу "Алгоритмы символьной обработки"

Итак, формальная обработка текстов связана с взглядом на последовательности f как текст в некотором алфавите tChar, f ∈ tWord=N→tChar или слова - подпоследовательности (интервалы) текста. Примеры - операции вставки, исключения и замены слов, обычные при редактировании текстов.

В реальности, мы в этом пункте *нигде* далее не используем даже какие-либо специфические особенности типа tChar как множества символов. На деле, здесь tChar - произвольный тип.

```
type
tChar=?;
tPosition=?;
tWord=?;
```

```
procedure Вставка(var ОсновнойТекст:tWord;ВставляемоеСлово: tWord; Позиция:tPosition);
```

Варианты - слово вставляется *после, начиная с* и *до* заданной позиции.

```
procedure Исключение
(var ОсновнойТекст:tWord; var Новое: tWord; НачалоИсключения,КонецИсключения :tPosition);
```

Варианты - те же, а также удаление подслова - процедура не порождает нового слова.

Все остальные операции формальной обработки текстов можно свести к операциям вставки и исключения. Так, например, операцию замены можно свести к исключению из текста одного слова и вставки другого, операцию порождения текста можно трактовать как вставку в пустой текст, а операцию уничтожения текста - как исключение (удаление) содержимого текста "из себя".

Другое дело - нужно ли это делать при заданной реализации текста. См. далее - при реализации текста списком мы в действительности выражаем вставку через порождение, удаление и копирование символов текста.

В свою очередь, операции вставки и исключения слов сводятся к кратной вставке и исключения символов.

Вставка и замена при представлении слов (псевдодинамическими) массивами.

```
type
tIndex=1..nMax; {максимальная длина текста+1}
tPosition=tIndex;
tWord=record   Content: array[tPosition] of tChar;   {содержимое слова/текста}
    Len: tPosition                {фактическая длина текста}
end;

procedure Insert {ВставкаПосле}
(var T {ОсновнойТекст} :tWord; W {ВставляемоеСлово}: tWord; P {Позиция}: tPosition);
var
i,          {текущая позиция в W}
```

```
k: tPosition ;          {=p+i, текущая позиция в T }
begin
i:=1;k:=p+1;
while (i<=W.Len) and (k<=nMax) do
begin {вставка i-го символа W}
    {сдвиг текста вправо на 1, начиная с позиции k}
    for j:= T.Len+1 downto k do T.Content[j+1]:= T.Content[j];
    T.Content[k]:=V.Content[i];
    inc(i);inc(k)
end; end;

procedure Exclude {Исключение}
(var T {ОсновнойТекст}:tWord; var V { Новое}: tWord; Start, Finish
{НачалоИсключения,КонецИсключения} : tPosition);
begin
var
k: tPosition; {текущая позиция в T, k ∈ [Start,Finish] }
begin
V.Len:=0;k:=Start;
while k<=Finish do
begin {исключение i-го символа W}
    inc(V.Len); V.Content[V.Len]:=T.Content[k];
    { удаление T.Content[k] - сдвиг текста влево на 1, начиная с позиции
k+1}
    for j:= k to T.Len do T.Content[j]:= T.Content[j+1];
    inc(k)
end; end;
```

Вставка и замена при представлении слов списочными структурами.

```
type
pComponent=^tComponent;
tComponent=record Content:  tChar;  {содержимое слова/текста}
          next:pComponent;   {ссылка на следующую компоненту}
end;
pPosition=pComponent;
pWord=pComponent; {ссылка на начало списка}

{область определения( предусловие): тест не пуст ≈  pT<>nil, p<>nil}
procedure Insert {ВставкаПосле}
(var  pT  {ОсновнойТекст}  :pWord;  pW  {ВставляемоеСлово}:  pWord;  P
{Позиция}:pPosition);
var
pn,                   {ссылка на новую компоненту}
pi,                   {ссылка на текущую позицию в W}
pk: pPosition;              {ссылка на текущую позиция в T }
begin
pi:=pW;pk:=P;
while pi<>nil do
begin {вставка i-го символа W}
      new(pn);                  {порождаем новую компоненту}
      pn^.Content:=pw.Content;       {с содержимым равным символу w[k]}
      pn^.next:=pk^.next;            {вставляем его перед T[i+1]}
      pk^.next:= pn;                 {и после T[i]}
      pw:=pw^.next;                  {переходим к следующим символам}
      pk:=pk^.next;
end; end;

procedure Exclude {Исключение}
```

```
(var pT {ОсновнойТекст}:pWord; var pV { Новое}: pWord; pStart, pFinish
{НачалоИсключения,КонецИсключения} : pPosition);
begin
var
pD,            {ссылка на уничтожаемый символ}
pN,            {ссылка на новый символ V}
pK: pPosition;      {ссылка на текущую позицию в T}
begin
new(pV);pV^.next:=nil; {вводим фиктивный/нулевой символ }
pLast:=pV;          {причина - определить ссылку на последний символ}
pK:=pStart;
while pK<>pFinish do
begin {исключение i-го символа W}
    {порождаем новый символ = T[k]}
    new(pN); pN^.Content:=pK^.Content; pN^.next:=nil;
    {делаем его последним в V}
    pLast^.Next:=pN; pLast:=pN;
    {удаляем текущий, переходим к следующему символу в T}
    pD:=pK; pK:=pK^.next; destroy(pD);
end;
{удаляем фиктивный/нулевой символ }
pD:=pV; pV:=pV^.next; destroy(pD)
end;
```

§ 15. Формальные вычисления.

См. преамбулу «Алгоритмы символьной обработки». В частности, там определено понятие формального вычисления как нахождение *текста* (обозначения) результата операции по *тексту* (обозначениям) аргументов, в некоторой фиксированной системе обозначений.

Очевидно, деление «обозначение(имя)/значение, синтаксис/семантика» является фундаментальным не только в программировании, но и в целом, для нашего мышления (см. **§ 1 «Основные понятия…»)**. Но программист обязан понимать и условность этого деления – в реальности, все компьютерные вычисления формальны, реализуясь в конечном счете в алгоритмах символьной обработки последовательностей битов (см. **§ 8 «Машинно-ориентированное программирование**»). Отсюда – очевидная важность рассматриваемой темы.

Рассмотрим ее на простом примере операции сложения натуральных чисел.

Задача. Найти запись (обозначение) суммы двух натуральных чисел по записям аргументов, не прибегая к преобразованиям типов. Обозначения чисел (в традиционной 10-ной системе счисления) представлены массивами.

```
type
tIndex=1..Len;
tCifer='0'..'9';
tNotation=array[tIndex] of tCifer;
tPair=record First,Second:tCifer end; {пара цифр}
{глобальный параметр - таблица сложения}
{PifagorTable: array[tCifer,tCifer] of tPair;}
procedure СложениеСтолбиком(Arg1,Arg2:tNotation; var Sum:tNotation;
Error:boolean);
```

```
{Error - ошибка переполнения}
var
        Um: '0'..'1'; {цифра "в уме"}
        Pair:tPair;
        i:tIndex;
begin
Um:='0';
for i:=1 to Len do
begin
        Pair:= PifagorTable[Arg1[i],Arg2[i]]; {пара цифр от '00' до '18'}
        if Um='0'
        then begin Sum[i]:=Pair.Second;Um:=Pair.First end
        else {Um=1}
                if Pair.Second='9' {особый случай}
        then { Pair.First='0' } Sum[i]:='0'
        else begin Sum[i]:=succ(Pair.Second);Um:=Pair.First end;
end;
Error:=Um='1'
end;
```

2. *Пример вычисления функции значения и обратной функции.*

Задача. Найти запись (обозначение) суммы двух натуральных чисел по записям аргументов, прибегая к преобразованиям типов. Обозначения чисел (в традиционной 10-ной системе счисления) представлены массивами (также, как в предыдущей задаче).

```
procedure ToValue(cArg:tNotation; var iResult:integer);
var a:0..9; {значение очередной цифры}
begin
iResult:=0;
```

{вычисление многочлена $\Sigma \{a_i \, 10^{Len-i} : i \in [1..Len]\}$ по схеме Горнера}

```
for i:=1 to Len do begin a:= ord(cArg[i]))-ord('0'); iResult:=iResult*10+a; end;
end;
procedure ToName(iArg:integer; var cResult:tNotation; Error:boolean);
var
c:tCifer; {очередная цифра десятичной записи}
a:0..9; {значение очередной цифры}
OrdZero:integer; {номер символа '0' в таблице символов}
begin
OrdZero:=Ord('0'); k:=1;
while (iArg<>0) and (k<=Len) do
begin
a:= iArg mod 10; iArg:=iArg div 10;
cResult[k]:=chr(OrdZero+a);k:=k+1
end;
Error:=k>Len;
while k<=Len do begin cResult[k]:='0';k:=k+1 end;
end;
procedure Sum(cArg1,cArg2:tNotation;var cResult:tNotation;Error:boolean);
var iArg1,iArg2:integer;
begin
toValue(cArg1,iArg1);
toValue(cArg2,iArg2);
ToName(iArg1+iArg2,cResult,Error);
end;
```

Литература

ОСНОВНАЯ ЛИТЕРАТУРА

1. Вирт Н. Систематическое программирование. Введение. М.: Мир, 1977. - 183 с.
2. Вирт Н. Алгоритмы+структуры данных = программы. М.: Мир, 1985. Алгоритмы и структуры данных. М: Мир, 1989. – 360 с.
3. Задачи для программирования по теме «Сортировка данных». Задачи для программирования по теме «Списки». Методические разработки под ред. В.С. Кугуракова. Казань, КГУ, 1987.
4. Марченко А.И. Программирование на языке *Object Pascal* 2.0. - К., ЮНИОР, 1998. - 304 с.

ДОПОЛНИТЕЛЬНАЯ ЛИТЕРАТУРА

1. С.А. Абрамов. Математические построения и программирование. М.: Наука ГРФМЛ, 1978.- 192.
2. А.В. Ахо, Дж.Д. Хопкрофт, Дж.Д. Ульман. Структуры данных и алгоритмы. М.: Вильямс, 2001. - 384 с.
3. А. Ахо, Дж. Хопкрофт, Дж. Ульман. Построение и анализ вычислительных алгоритмов. М.: Мир, 1979. - 536 с.
4. Гудман С., Хидетниеми С. Введение в разработку и анализ алгоритмов. М.: Мир, 1981.
5. У. Дал, Э. Дейкстра, К. Хоор. Структурное программирование. М.: Мир, 1975. - 248 с.
6. М. Зелковиц и др. Принципы разработки программного обеспечения. М.: Мир, 1982. - 368 с.
7. Зиглер К. Методы проектирования программных систем. Мир, 1985.
8. К. Кормен, Ч. Лейзерсон, Р. Ривест. Алгоритмы: построение и анализ. М.: НЦНМО, 2001. - 960 с.
9. С. Лавров. Программирование. Математические основы, средства, теория. СПб.: БХВ-Петербург, 2001. - 320 с.
10. Б. Мейер, К. Бодуен. Методы программирования. т1-2. М.: Мир, 1982.
11. Структуры и базы данных. М., Мир. 1986.- 198 с.
12. Холл П. Вычислительные структуры. Введение в нечисленное программирование. М.: Мир, 1978.