УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«Брестский государственный университет им. А.С. Пушкина»

ПРОГРАММИРОВАНИЕ

Курс лекций по разделу

«Алгоритмизация с визуализацией средствами Р-схем»

для студентов физико-математического факультета

Брест
БрГУ имени А.С. Пушкина
2017

Учреждение образования
«Брестский государственный университет имени А.С. Пушкина»

Н.В. Силаев, А.И. Серый, З.Н. Серая

КУРС ЛЕКЦИЙ

«АЛГОРИТМИЗАЦИЯ С ВИЗУАЛИЗАЦИЕЙ СРЕДСТВАМИ Р-СХЕМ»

ОТД. ПМ И ЭК (1 СЕМ.)

Брест 2017

Оглавление

РАЗДЕЛ 1 ОСНОВЫ АЛГОРИТМИЗАЦИИ 5

Глава 1 Визуализация алгоритмов средствами Р-схем 5
§1 История и цели разработки 5
§2 Базис ВТР 8
§3 Классификация процессов и Р-схемы 11
3.1 Линейные процессы и варианты их представления 11
3.2 Ветвящиеся процессы и варианты их представления 12
3.3 Циклические процессы и варианты их представления 13
§4. Варианты построения вложенных Р-схем 19
4.1 Построение композиций на основе структуры «Ветвление» 19
4.2 Построение композиций на основе структуры «Цикл» 26
§5. Классификация простых циклических процессов 32
5.1 Арифметические циклы 33
5.2 Итерационные циклы 40
5.3 Циклы с Контролем за Монотонной Величиной 46
5.4 Циклы с Внутренним Контролем 49
§6. Классификация циклических процессов обработки массивов 54
6.1 Задачи на заполнение табличных величин 56
6.2 Задачи на анализ табличных величин 57
6.3 Задачи на поиск в табличных величинах 58
6.4 Задачи на перестановки в табличных величинах 59
§7. Классификация циклических процессов обработки строковых величин 67
§8. Базовые схемы построения циклов 76
8.1 Предварительные замечания 76
8.2 Рекурсивная схема построения циклов 77
8.2.1 Определения и пример 77
8.2.2 Требования, предъявляемые к рекурсивным алгоритмам 79
8.2.3 Общая методика построения рекурсивного алгоритма 79
8.2.4 Достоинства и недостатки схемы 82
8.2.5 Семантика обработки рекурсивных алгоритмов. Формуляры 83
8.2.6 Примеры написания простых рекурсивных алгоритмов 88
§9. О структурности в программировании 96
9.1 Совершенствование методов программирования: 96
9.2 Наиболее популярные технологии 96
9.3 Структура представления данных состоит из двух частей: 97
9.4 Распределение памяти 98
9.5 Способы распределения памяти 103
9.6 Виды распределения памяти 105
9.6.1 Непрерывное представление объектов в памяти 105

9.6.2 Разрывное представление объектов в памяти 106
9.6.2 Списочное представление объектов в памяти 107
9.6.3 Иерархически организованные объекты 110
9.6.4 Структура внешней памяти 110
9.7 Структура данных ООП 111
§10. Основная задача программирования. Понятие исполнителя 112
10.1 Определения и соглашения: 112
10.2 Каково назначение аппарата И? 113
Приложение 117
Ответы к заданиям 122
Экспресс проверка усвоения некоторых понятий 124
Литература 125

Раздел 1 Основы алгоритмизации

Глава 1 Визуализация алгоритмов средствами Р-схем

К оглавлению *К следующему параграфу*

§1 История и цели разработки

- Программист пишет программу, компьютер ее выполняет. Однако не все до конца осознают тот факт, что прежде чем писать программу, программисту ***необходимо*** сформировать каким-то образом алгоритм – суть решаемой задачи – независимо от ее сложности и способа оформления.
- С появлением персональных компьютеров и высокоуровневых языков программирования основное большинство программистов (в особенности начинающих и малоопытных) предпочитают *сразу приступать к написанию кода*.
- Подобный подход, как правило, приводит к тому, что процесс написания программы (между прочим, это тоже *способ описания алгоритма, но предельно детализированный* «под возможности» компьютера) напоминает «блуждание ежика в тумане», особенно у непрофессиональных программистов (учеников и студентов).
- Очень эффективным средством описания алгоритмов, и на наш взгляд, вполне доступным для учащейся аудитории практически любого уровня, являются Р-схемы.
- Более подробно это средство будет освещаться ниже, а сейчас отметим наиболее привлекательные на наш взгляд черты как самих схем, так и среды их разработки:
  - наглядность алгоритмов, представленных Р-схемой(-ами);
  - возможность просмотра исключительно структуры схем, исключением нагрузок;
  - возможность оформления схем любого уровня сложности и любой тематики[1];
  - простота внесения управляющих структур в Р-схему и простота модификации построенных схем;
  - естественность оформления алгоритма и простота реализации его декомпозиции на основе уже построенной Р-схемы более высокого уровня[2];

[1] В данном случае нас привлекает в первую очередь учебная тематика: исполнители, классические задачи на построение ветвлений, простых циклов разного типа, обработки стандартных языковых структур данных (массивов, строк, записей и т.п.).
[2] Именно это обстоятельство непринужденно прививает навыки нисходящей разработки алгоритмов.

- возможность простого документирования алгоритмов в самой среде разработки Р-схем;
- удобная среда разработки Р-схем (R-Editor) с интуитивно понятным интерфейсом и устойчивой работой.

<Общий вид среды. Папка 01 Начальное состояние\01 Первый вход>;

- возможность легкой конвертации разработанного семейства иерархически связанных Р-схем алгоритма во множество функций С++-кода[3].

▪ Не останавливаясь подробно на истории возникновения Р-схем (в начальных обозначениях R-схем), заметим лишь, что они являются составляющей частью R-технологии [1-2] (RTK ЕС, RTK БЭСМ-6) [3] и были предложены И.В. Вельбицким.

▪ «*R-технология предлагает единый математический аппарат для всего технологического комплекса производства программных средств:* ***проектирования***[4]*, кодирования, отладки, документирования и эксплуатации*».[5]

▪ Данное средство уникально тем, что оно поддерживается государственным стандартом ГОСТ 19.005-85 и является, пожалуй, единственной отечественной технологией программирования, на которую в 1988 был получен международный стандарт ISO 8631 [4], и которая была высоко оценена зарубежными специалистами [5].

▪ С появлением графических дисплеев и персональных ЭВМ, Р-схемы обрели «полное дыхание» и оказались жизненно востребованными средствами визуальной технологии программирования.

> *Вот что об этом пишет И.В. Вельбицкий: «…Визуальная технология программирования нового поколения (ВТР) использует так называемые Р-схемы. Р-схемы – это нагруженные по дугам ориентированные графы. Основное отличие ВТР – простота и единая на всем жизненном цикле графическая форма записи программ, обеспечивающая новые принципы проектирования, отладки, наглядную и компактную (более чем на порядок) запись по сравнению с записью в существующих языках программирования (ЯП). Ввиду своей простоты доступна широкой массе специалистов, а не только программистам.... В настоящее время графическое программирование ВТР реализовано в среде Qt-Criotor C++ (www.glushkov.org). Это позволило вполне обоснованно провести анализ преимуществ ВТР в сравнении с традиционными технологиями, использующими современные ЯП, UML и ООП…*
>
> *В технологии ВТР нового поколения из программирования исключаются операторы перехода, выбора и цикла типа goto, for, while, а также метки и скобки типа begin-end, {-} и т.д.*

[3] Подобная конвертация практически осуществима и в исходные кода других высокоуровневых языков, за счет внесения в код результатов конвертаций незначительных изменений «косметического характера».
[4] Выделено мною, автором данных материалов.
[5] Там же, стр. 5.

Вместо них предлагается использовать теорию нагруженных по дугам ориентированных графов из классической математики. В результате программирование приобретает единый (один-единственный!) графический эквивалент (ядро, графическую оболочку) для всех ЯП. В новом программировании инструмент (графическое ядро) развивается, подстраивается под решаемую задачу и ее исполнителей, а не задача трансформируется под инструмент (операторы ЯП), как в традиционном программировании. Это значительно упрощает и решает существующие проблемы традиционного программирования, включая ООП» [6].

К оглавлению К следующему параграфу К предыдущему параграфу

§2 Базис ВТР

- Графическая оболочка алгоритма в предлагаемом средстве (языке) нового поколения строится на основании схем.
- Элементарная схема данной ВТР называется ***Р-схемой***. Она состоит из *одной горизонтальной дуги* между *двумя вершинами*.

Рис. 2. 1 Вид элементарной Р-схемы <Док. 02>

- Дуга имеет направление только *вправо* или *влево*.
- Из вершины Р-схемы может выходить *любое число дуг* вправо и/или влево.

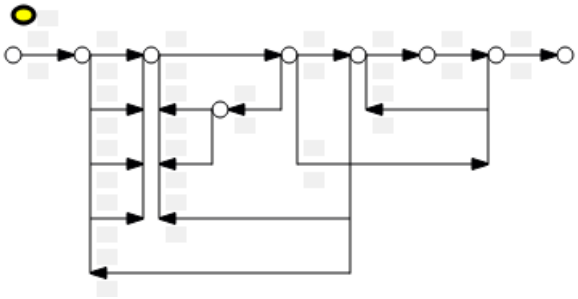


Рис. 2. 2 Пример Р-схемы с ненагруженными дугами <Док. 03>*

- Р-схема рис. 2.2 состоит из четырех циклов.
- Заметим, что исходящие из вершин дуги просматриваются (читаются, понимаются, анализируются, и выполняются) *последовательно*
  - *сверху вниз* и
  - от *вершины к вершине по соответствующей стрелке дуги*,
  - начиная *от первой слева вершины* и
  - кончая *последней вершиной справа*.
- В Р-схемах *вертикальные дуги* являются ***вспомогательными***, они соединяют вершины с *горизонтальными дугами*, которые являются ***основными.***
- Только *основные дуги нагружаются информацией*.
- Другой геометрический объект Р-схемы – ***вершина*** – *не имеет имени* и *задает состояние программы* или процесса ее разработки.

- Ввод графа в среде редактор схем *REditor* осуществляется только вводом горизонтальных дуг около выделенной одной или двух вершин[6]. При этом первая дуга Р-схемы и надписи на ней (Рис.1.1) всегда рисуются автоматически.
- В Р-схемах на дуге сверху пишется ***Условие*** прохождения по дуге, а снизу выполняемые при этом ***Действия*** (Рис.2.1)[7].
- На запись ***Условий*** и ***Действий*** не накладывается никаких ограничений – они могут быть записаны/не записаны на любом языке: русском, английском, китайском, математическом, языке программирования и т.д. в одну или несколько строк.

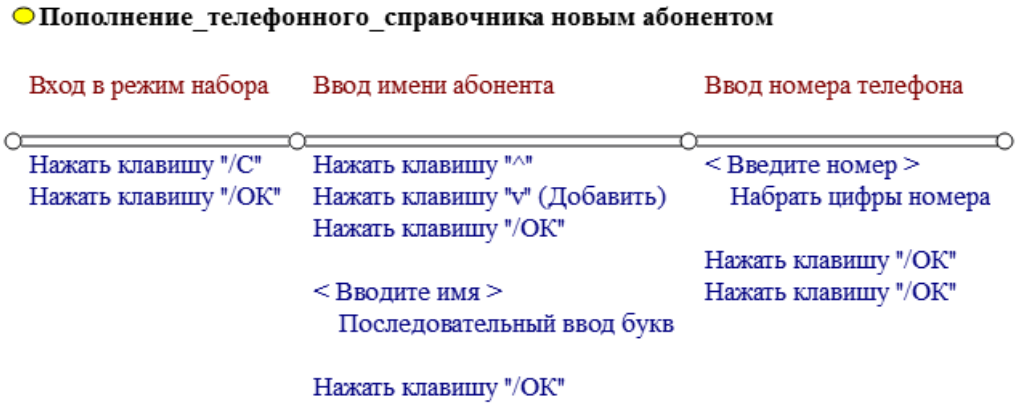


Рис. 2. 3 Алгоритм корректировки номера в справочнике радиотелефона <Док. 04.01>

Рис. 2. 4 Фрагмент описания синтаксиса понятия Буква на C# <Док. 04.02>

[6] Например для ввода Р*-схемы рис.2, состоящей из 17 дуг, необходимо 16 нажатий клавиш мыши.
[7] ***Условие*** и/или ***Действия*** могут отсутствовать.

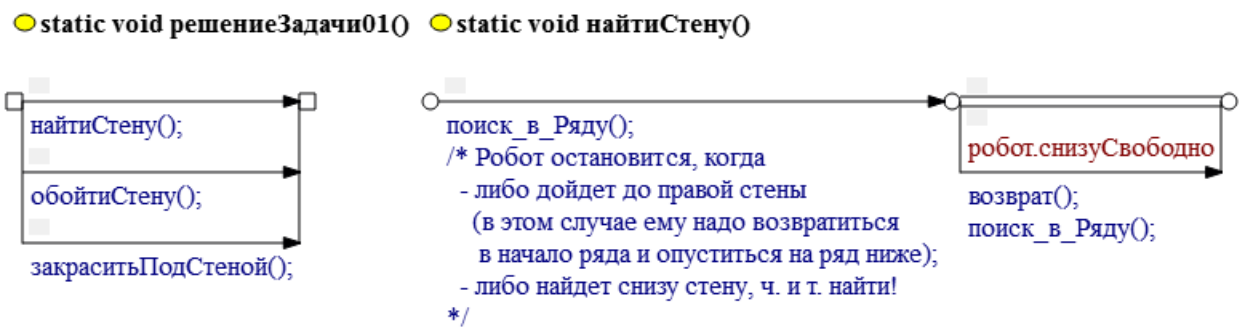


Рис. 2. 5 Фрагменты алгоритма для исполнителя Робот на языке C#<Док. 04.03>

Рис. 2. 6 Детализированная и «свернутая» Р-схемы решения задачи на ветвление <Док. 04.04>

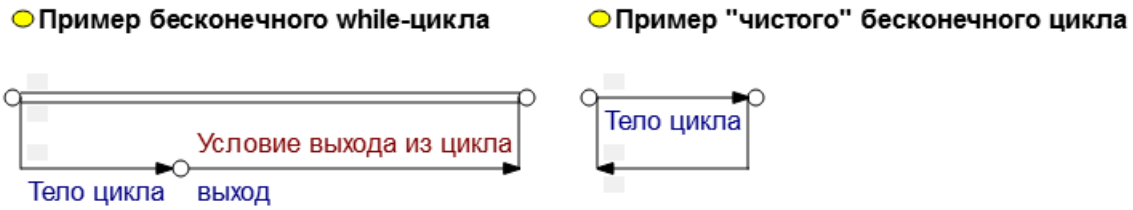


Рис. 2. 7 Варианты построения бесконечных циклов с/без контроля за выходом <Док. 04.05>

К оглавлению К следующему параграфу К предыдущему параграфу

§3 Классификация процессов и Р-схемы[8]

- Традиционно множество алгоритмов, реализуемых средствами программирования, классифицируют, как показано на следующей схеме:

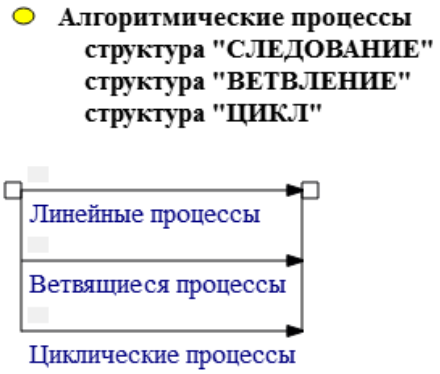


Рис. 3. 1 Классификация алгоритмических процессов

- Для *линейного процесса* характерна справедливость

  Свойства 1: Выполняются все команды процесса и

  Свойства 2: Каждая команда процесса выполняется точно один раз.

3.1 Линейные процессы и варианты их представления

- Средствами Р-схем такие процессы могут быть представлены в следующих вариантах:

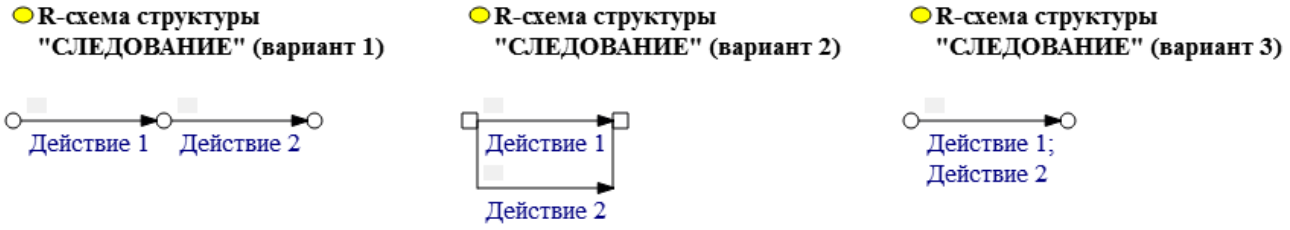


Рис. 3. 2 Варианты оформления линейных алгоритмических процессов

- Всем вариантам характерно отсутствие указания ***Условия*** над дугой (дугами 1 вариант).
- Если же ***Условие*** указано, то мы имеем дело либо с *ветвящимся,* либо с *циклическим* процессами.
- Заметим, что при использовании *REditor*, интегрированного с языком программирования, например с C++, допустим вариант «расширения» возможностей организации фрагмента линейного алгоритма. Так, если ***Условие*** начинается с *ключевого слова*, то это ***условие является всегда***

[8] Ознакомиться с иллюстрирующими Р-схемами данного и следующего параграфов можно в документе 02, папки 01, «01 Базовые структуры структурной технологии» и 02 – «02 Возможные взаимные расположения структур» <Док. 02.01-02>

истинной логической константой *Р-схемы* и *выполняется особым способом*, оговоренном при определении графической оболочки.

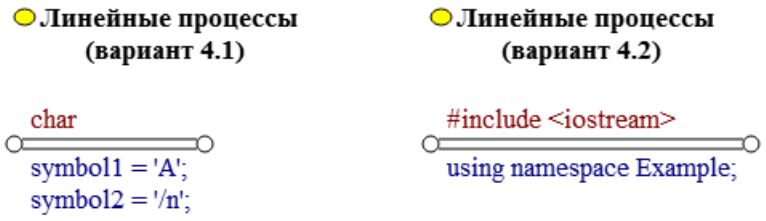


Рис. 3. 3 Вариант оформления фрагмента линейного алгоритма с ключевым словом <Назад>

- Как правило, *логическая константа* присоединяется к своему ***Действию***, образуя единый результирующий код.
- Такая возможность обеспечивает эффективное графическое задание описаний данных и метаданных.

3.2 Ветвящиеся процессы и варианты их представления

- *Ветвящийся процесс* характерен тем, что в нём *нарушается* ***свойство 1***.
- За счет чего это происходит? «Виной» тому – наличие ***Условия*** над дугой.
- Если ***Условие*** **истинно**, то выполняются ***Действия*** под этой дугой и происходит переход по стрелке к следующей вершине.
- Если ***Условие*** **ложно**, то ***Действия*** под дугой не выполняются и рассматривается *следующая сверху вниз выходящая из этой вершины дуга*, а если ее нет, то дуга, следующая за вершиной, на которую она (дуга с последним ложным условием) указывает.
- Простые варианты Р-схемы для этого вида процессов таковы:

Рис. 3. 4 Варианты оформления ветвящихся процессов

- В данном случае

- вариант 1 иллюстрирует наличие обеих альтернатив – *полное ветвление*;
- варианты 2, 3.1 и 3.2 – наличие лишь одной альтернативы – *укороченное ветвление*, в случае 2 – альтернативы «то», а в остальных двух – только альтернативы «иначе». При этом вариант 3.1 менее желателен для использования (из-за «пустоты» ***Действия***, соответствующего альтернативе «то»). Поэтому вариант 3.2 предлагается как рекомендуемый для использования в случаях, когда по контексту задачи альтернатива «то» должна отображаться «пустым» действием.

3.3 Циклические процессы и варианты их представления

- *Циклический процесс* характерен тем, что в нём нарушается ***свойство 2***.
- В данном случае заметим, что циклические процессы характеризуются б**о**льшим многообразием, чем два вида предыдущих процессов.
- Дело в том, что ***Условие***, *задающее необходимость неоднократного* выполнения группы действий (*тела цикла*) может быть указано
  - либо в начале процесса – *цикл с предусловием.* В высокоуровневых языках программирования это, как правило, оператор ***while***;
  - либо в конце процесса – *цикл с постусловием.* В высокоуровневых языках программирования это, как правило, оператор ***do-while*** или ***repeat-until***;
  - либо внутри процесса – *цикл с выходом из него по условию.* В высокоуровневых языках программирования это оператор цикла, организующий бесконечное или конечное (длительное) число повторений (например, ***while*** true…), тело которого содержит проверку некоторого условия, значение истинности которого означает *выход за пределы области цикла*;
  - либо «спрятано» в семантике оператора цикла. В высокоуровневых языках программирования это, как правило, оператор цикла *с параметром* ***for.***
- Описанные варианты этого вида процессов средствами Р-схем могут описываться так:

- ***Цикл с предусловием:***

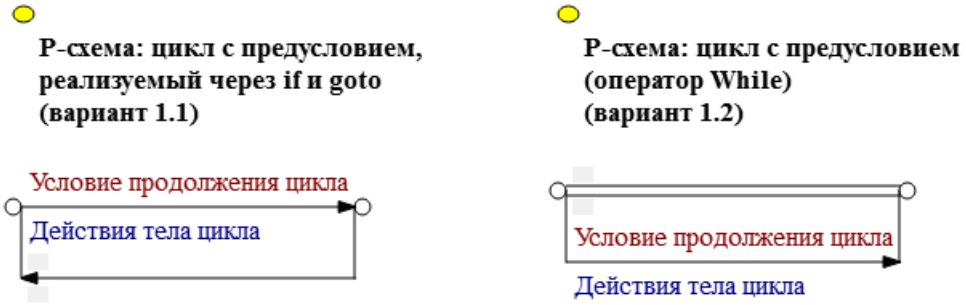


Рис. 3. 5 Варианты оформления циклических процессов предусловием

- Варианты 1.1 и 1.2 иллюстрируют цикл *с предусловием.*
- В первом случае строится полная конструкция оператора ***while*** согласно синтаксиса ***C***-подобных языков.
- Во втором случае строится семантический эквивалент оператору цикла ***while,*** но с использованием операторов ***if*** и ***goto:***

```
P-схема: цикл с предусловием,
реализуемый через if и goto
(вариант 1.1)
{
    _l15:
    if(Условие продолжения цикла){
        Действия тела цикла
    }else{
        goto _l16e;
    }
    goto _l15;
    _l16e:;
}

P-схема: цикл с предусловием
(оператор While)
(вариант 1.2)
{
    while(Условие продолжения цикла){
        Действия тела цикла
    }
}
```

Рис. 3. 6 Результаты трансляции P-схем рис. 3.5

- ***Цикл с постусловием:***

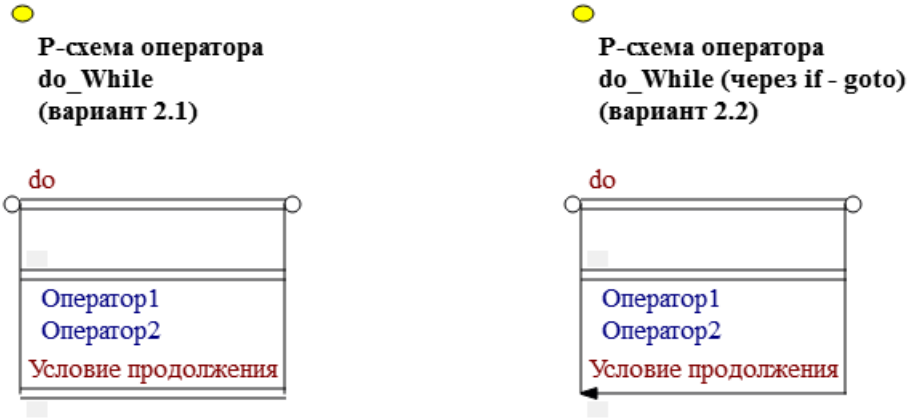


Рис. 3. 7 Варианты оформления циклических процессов с постусловием

- Варианты 2.1 и 2.2 иллюстрируют цикл *с предусловием.*
- В результате трансляции приведенных P-схем получаем:

```
P-схема оператора
do_While
(вариант 2.1)
{
    do{
        Оператор1
        Оператор2
      while(Условие продолжения){
      }
    }
     ;
}
```

```
P-схема оператора
do_While (через if - goto)
(вариант 2.2)
{
    _l11:
    do{
        Оператор1
        Оператор2
    }
     ;
    if(Условие продолжения){
      goto _l11;
    }
}
```

Рис. 3. 8 Результаты трансляции P-схем рисунка 3.7

- ***Цикл с параметрическим управлением:***

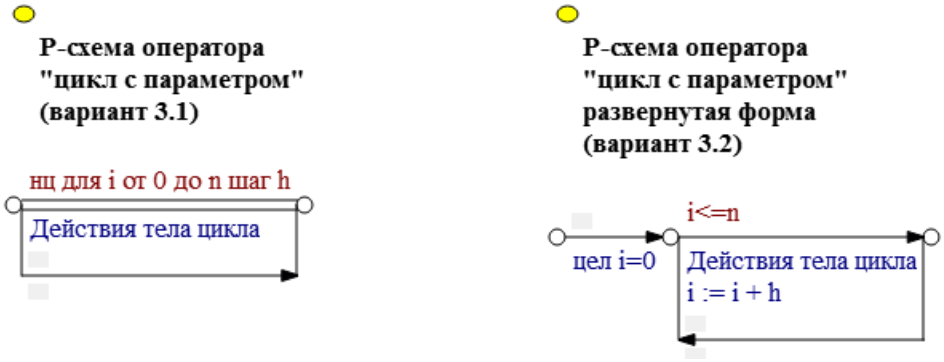


Рис. 3. 9 Р-схемы циклических процессов с «параметрическим» управлением

- Варианты 3.1 и 3.2 иллюстрируют цикл *с параметром.*
- В результате трансляции приведенных Р-схем получаем:

```
Р-схема оператора
"цикл с параметром"
(вариант 3.1)
{
    нц для i от 0 до n шаг h{
        Действия тела цикла
    }
}
Р-схема оператора
"цикл с параметром"
развернутая форма
(вариант 3.2)
{
    цел i=0
    _l29:
    if(i<=n){
        Действия тела цикла
        i := i + h
    }else{
        goto _l30e;
    }
    goto _l29;
    _l30e:;
}
```

Рис. 3. 10 Результаты трансляции Р-схем рисунка 3.9

- ***Цикл с дополнительным внутренним контролем:***

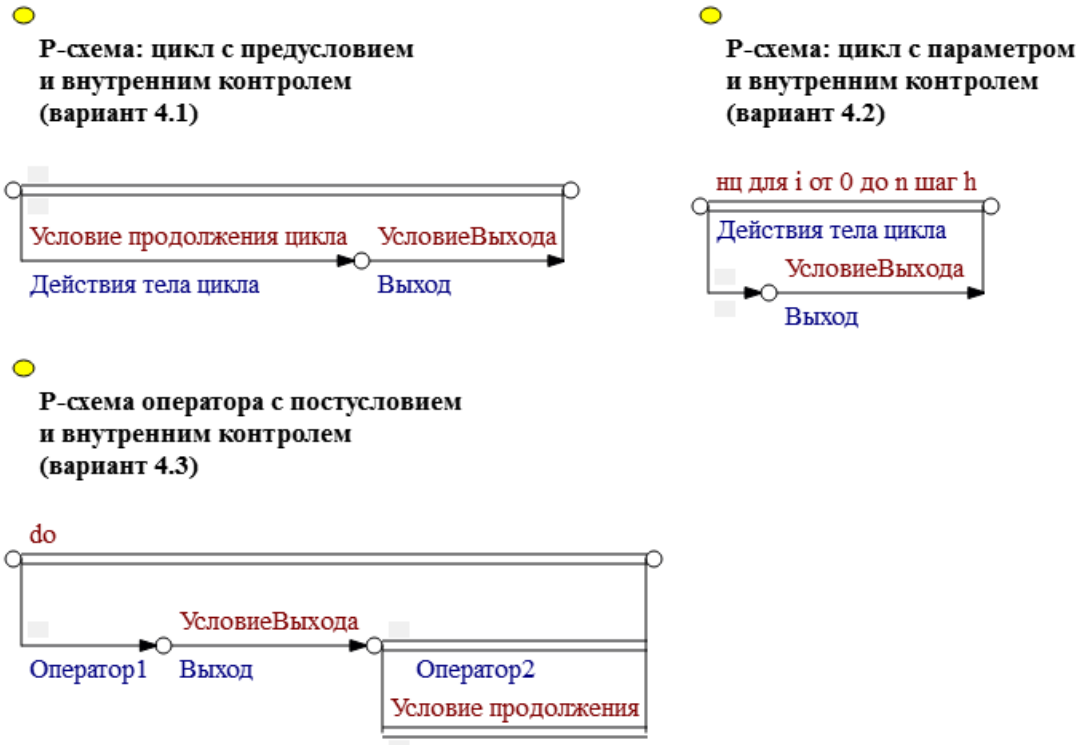


Рис. 3. 11 Вариант оформления циклических процессов с «внутренним» условием

- Вариант 4.1 иллюстрирует цикл *с предусловием* и *внутренним контролем*.
- Вариант 4.2 иллюстрирует цикл *с параметром* и *внутренним контролем.*
- Вариант 4.3 иллюстрирует цикл *с постусловием* и *внутренним контролем*.
- В результате трансляции приведенных Р-схем получаем:

```
Р-схема: цикл с предусловием
и внутренним контролем
(вариант 4.1)
{
    while(Условие продолжения цикла){
        Действия тела цикла
        if(УсловиеВыхода){
            Выход
        }
    }
}
```

```
Р-схема: цикл с параметром
и внутренним контролем
(вариант 4.2)
{
    нц для i от 0 до n шаг h{
        Действия тела цикла
        if(УсловиеВыхода){
            Выход
        }
    }
}
  Р-схема оператора с постусловием
  и внутренним контролем
  (вариант 4.3)
  {
      do{
          Оператор1
          if(УсловиеВыхода){
              Выход
          }
            Оператор2
          while(Условие продолжения){
          }
      }
  }
```

Рис. 3. 12 Результаты трансляции Р-схем рисунка 3.11

- Заметим, что если над специальной дугой рис. 3.9 (вариант 3.1) указать ключевое слово ***foreach***, то получим полный эквивалент оператору перебора языка C#.

К оглавлению К следующему параграфу К предыдущему параграфу

§4. Варианты построения вложенных Р-схем

- При построении алгоритмов нередко возникает необходимость вложения одних процессов в другие, т.е. построения композиции основных структур, о которых говорилось выше.
- Проиллюстрируем гибкость средств редактирования Р-схем при построении различных вариантов композиций основных процессов.

4.1 Построение композиций на основе структуры «Ветвление»

- Заметим, что данная ситуация построения композиции в практике программирования является самой «коварной» вследствие «уязвимости» по факту появления ошибок в ходе непосредственного кодирования.
- ***Вложение структуры Следование в альтернативу «то» структуры Ветвление***

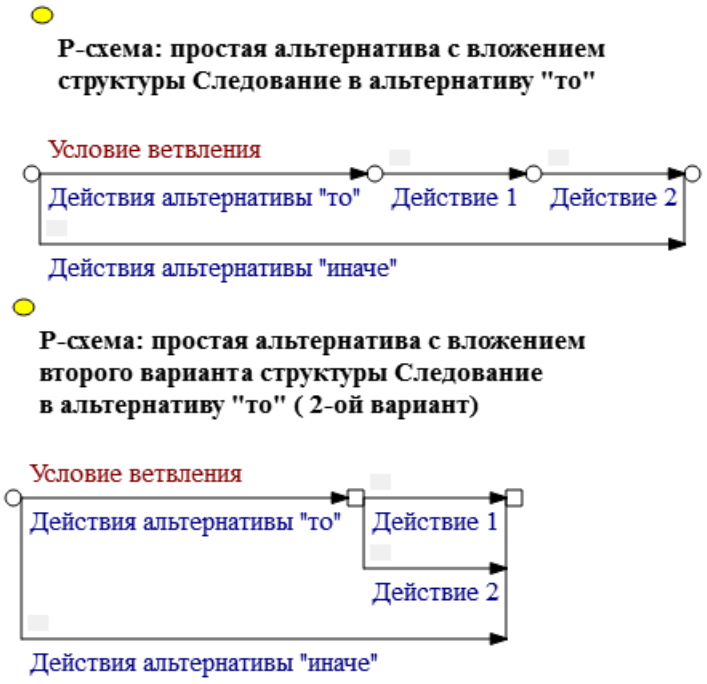


Рис. 4. 1 Варианты вложения фрагментов линейных процессов в альтернативу «то»

- Результат трансляции обеих Р-схем одинаков:

```
Р-схема: простая альтернатива с вложением
структуры Следование в альтернативу "то"
{
    if(Условие ветвления){
        Действия альтернативы "то"
        Действие 1
        Действие 2
    }else{
        Действия альтернативы "иначе"
    }
}
```

Рис. 4. 2 Результаты трансляции Р-схем рисунка 4.1

- ***Вложение структуры Следование в альтернативу «иначе» структуры Ветвление***

Рис. 4. 3 Варианты вложения фрагментов линейных процессов в альтернативу «иначе»

- Результат трансляции обеих Р-схем одинаков:

```
Р-схема: простая альтернатива с вложением
структуры Следование в альтернативу "иначе"
{
    if(Условие ветвления){
        Действия альтернативы "то"
    }else{
        Действия альтернативы "иначе"
        Действие 1
        Действие 1
    }
}
```

Рис. 4. 4 Результаты трансляции Р-схем рисунка 4.3

- ***Обобщенный вариант включения кода линейных процессов в обе альтернативы:***

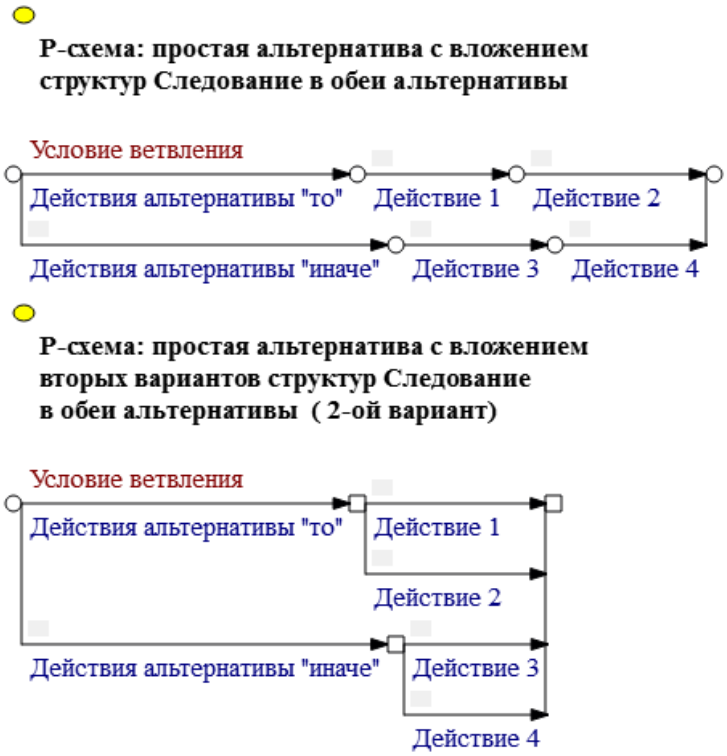


Рис. 4. 5 Несколько фрагментов линейных процессов в обеих альтернативах ветвления

- Результат трансляции обеих Р-схем одинаков:

```
Р-схема: простая альтернатива с вложением
структур Следование в обеи альтернативы
{
    if(Условие ветвления){
        Действия альтернативы "то"
        Действие 1
        Действие 2
    }else{
        Действия альтернативы "иначе"
        Действие 3
        Действие 4
    }
}
```

Рис. 4. 6 Результаты трансляции Р-схем рисунка 4.5

- На очередных иллюстрациях мы демонстрируем гибкость возможностей Р-схем в оформлении семантически эквивалентных алгоритмов.
- Надо обратить внимание на ту особенность построения Р-схемы, что основная структура *Ветвления* может не разрастаться «вширь», но конфигурироваться «по высоте»:

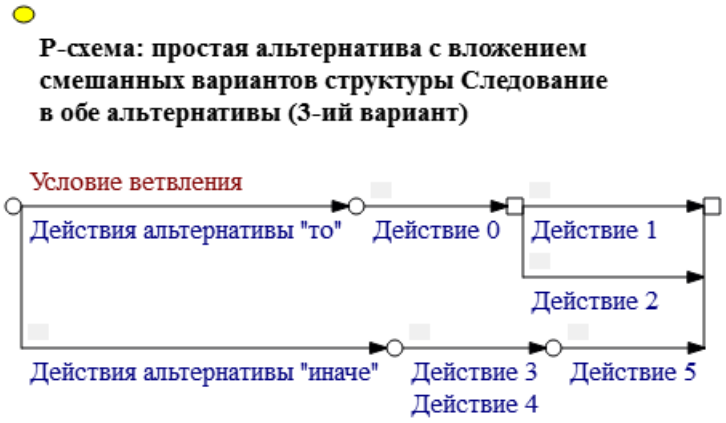


Рис. 4. 7 Разные варианты включения структуры Следование в обе альтернативы

- Результат трансляции Р-схемы по-прежнему удовлетворяет логике включения:

```
Р-схема: простая альтернатива с вложением
смешанных вариантов структуры Следование
в обе альтернативы (3-ий вариант)
{
    if(Условие ветвления){
        Действия альтернативы "то"
        Действие 0
        Действие 1
        Действие 2
    }else{
        Действия альтернативы "иначе"
        Действие 3
        Действие 4
        Действие 5
    }
}
```

Рис. 4. 8 Результаты трансляции Р-схем рисунка 4.7

- Не вдаваясь в подробности, аналогично вышеприведенным вариантам вложения структуры *Следования* в структуру *Ветвление*, приведем *Р-схему* ***вложения структур Ветвление в структуру Ветвление***:

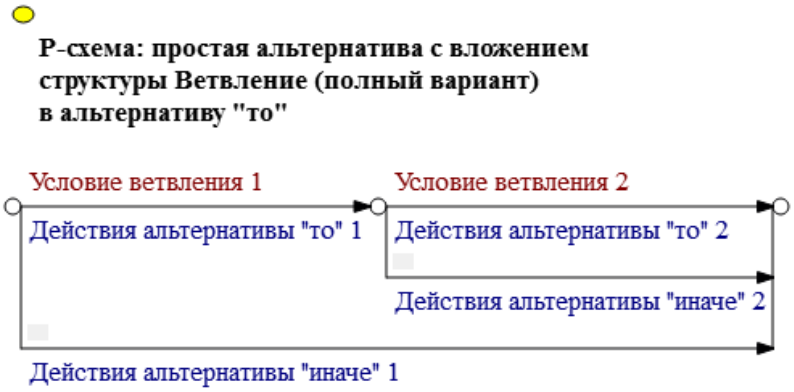


Рис. 4. 9 Вложение Ветвления в альтернативу «то» внешней структуры Ветвление

```
Р-схема: простая альтернатива с вложением
структуры Ветвление (полный вариант)
в альтернативу "то"
{
    if(Условие ветвления 1){
        Действия альтернативы "то"1
        if(Условие ветвления 2){
            Действия альтернативы "то"2
        }else{
            Действия альтернативы "иначе"2
        }
    }else{
        Действия альтернативы "иначе"1
    }
}
```

Рис. 4. 10 Результат трансляции Р-схемы рисунка 4.9

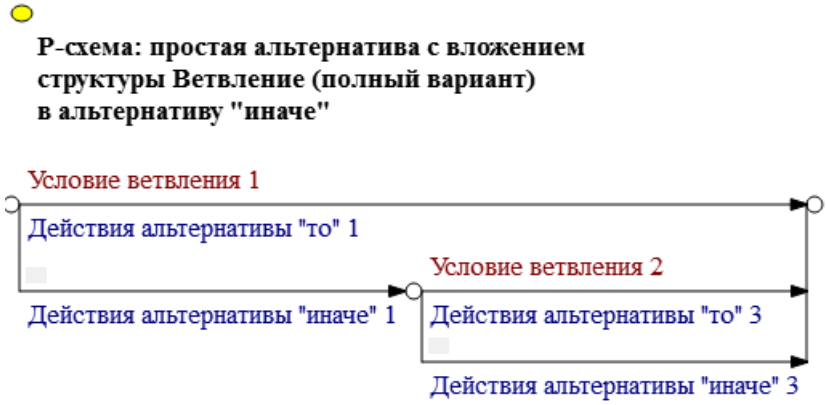


Рис. 4. 11 Вложение Ветвления в альтернативу «иначе» внешней структуры Ветвление

```
Р-схема: простая альтернатива с вложением
структуры Ветвление (полный вариант)
в альтернативу "иначе"
{
    if(Условие ветвления 1){
        Действия альтернативы "то"1
    }else{
        Действия альтернативы "иначе"1
        if(Условие ветвления 2){
            Действия альтернативы "то"3
        }else{
            Действия альтернативы "иначе"3
        }
    }
}
```

Рис. 4. 12 Результат трансляции Р-схемы рисунка 4.11

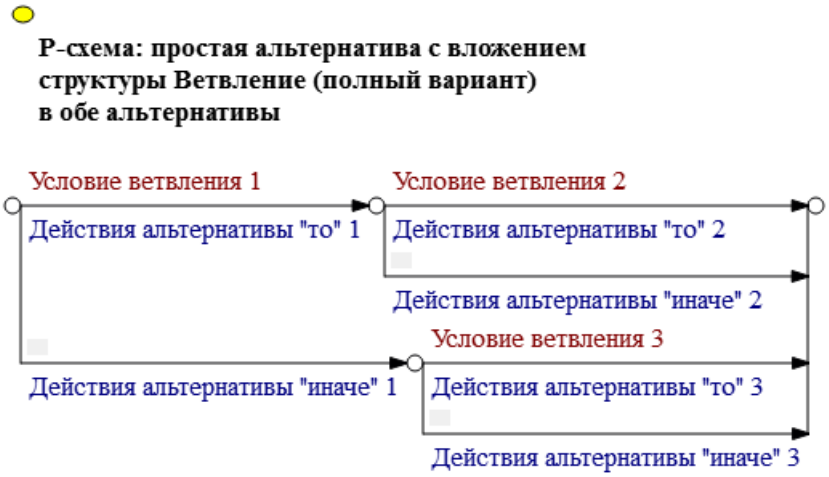
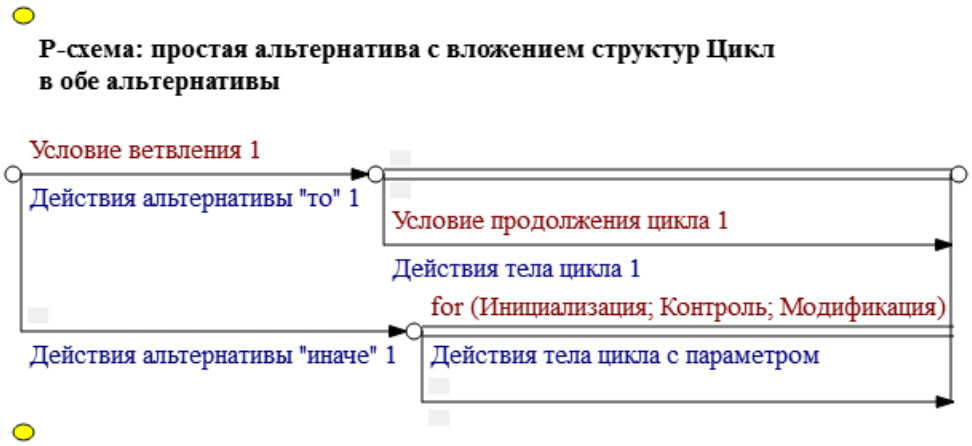


Рис. 4. 13 Вложение двух Ветвлений в обе альтернативы внешней структуры Ветвление

```
Р-схема: простая альтернатива с вложением
структуры Ветвление (полный вариант)
в обе альтернативы
{
    if(Условие ветвления 1){
        Действия альтернативы "то"1
        if(Условие ветвления 2){
            Действия альтернативы "то"2
        }else{
            Действия альтернативы "иначе"2
        }
    }else{
        Действия альтернативы "иначе"1
        if(Условие ветвления 3){
            Действия альтернативы "то"3
        }else{
            Действия альтернативы "иначе"3
        }
    }
}
```

Рис. 4. 14 Результат трансляции Р-схемы рисунка 4.13

- ***Вложение структуры Цикл в структуру Ветвление***:



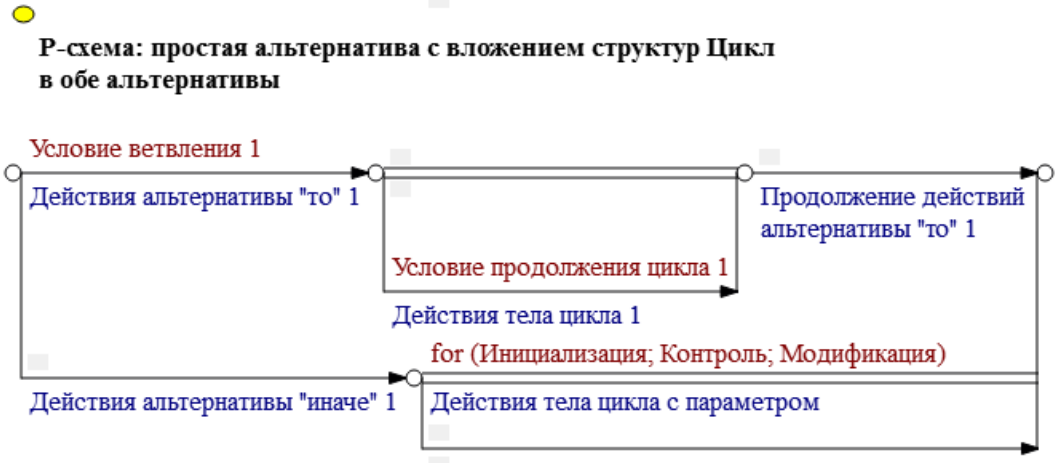


Рис. 4. 15 Вложение структур Цикл разного вида в структуру Ветвление

```
Р-схема: простая альтернатива с вложением структур Цикл
в обе альтернативы
{
    if(Условие ветвления 1){
        Действия альтернативы "то"1
        while(Условие продолжения цикла 1){
            Действия тела цикла 1
        }
    }else{
        Действия альтернативы "иначе"1
        for (Инициализация; Контроль; Модификация){
            Действия тела цикла с параметром
        }
    }
}

Р-схема: простая альтернатива с вложением структур Цикл
в обе альтернативы
{
    if(Условие ветвления 1){
        Действия альтернативы "то"1
        while(Условие продолжения цикла 1){
            Действия тела цикла 1
        }
        Продолжение действий альтернативы "то"1
    }else{
        Действия альтернативы "иначе"1
        for (Инициализация; Контроль; Модификация){
            Действия тела цикла с параметром
        }
    }
}
```

Рис. 4. 16 Результаты трансляции композиционных Р-схем «Ветвление - Циклы»

4.2 Построение композиций на основе структуры «Цикл»

- В данном пункте мы ограничимся рассмотрением, в качестве основы, лишь структуры *Цикл с предусловием*, т.к. для остальных видов цикла ситуация будет совершенно аналогичной.
- Поскольку вложение структуры *Следование* в структуру *Цикл с постусловием* особого интереса не представляет[9], то здесь мы начнем с вложения структуры *Ветвление* в структуру *Цикл с постусловием*.

9 Дело в том, что само *Тело цикла* можно рассматривать как структуру *Следования*. Композиция же нескольких структур *Следования* – есть структура *Следования*.

- ***Вложение структуры Ветвления в структуру Цикл с постусловием:***

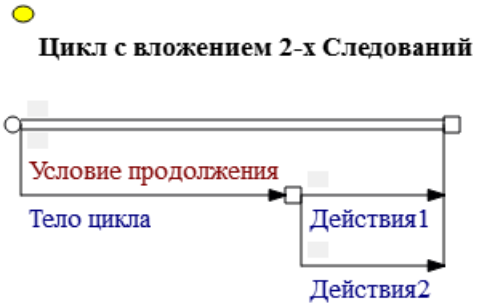


Рис. 4. 17 Р-схема композиции структур «Цикл-пока – Следование»

```
Цикл с вложением 2-х Следований
{
    while(Условие продолжения){
        Тело цикла
        Действия1
        Действия2
    }
}
```

Рис. 4. 18 Результат трансляции Р-схемы рисунка 4.17

- ***Вложение структуры Ветвления в структуру Цикл с постусловием:***

Рис. 4. 19 Р-схема композиции структур «Цикл-пока – Ветвление»

```
Р-схема: композиция структуры Цикл с предусловием
со структурой Ветвление
{
    while(Условие продолжения){
        Тело цикла
        if( Условие вложенного Ветвления){
            Действия альтернативы "то"
        }else{
            Действия альтернативы "иначе"
        }
    }
}

Р-схема: композиция структуры Цикл с предусловием
со структурой Ветвление
{
    while(Условие продолжения){
        Тело цикла (начало)
        if( Условие вложенного Ветвления){
            Действия альтернативы "то"
        }else{
            Действия альтернативы "иначе"
        }
        Тело цикла (окончание)
    }
}
```

Рис. 4. 20 Результат трансляции Р-схемы рисунка 4.19

- Следовательно, средства Р-схем в данном случае вполне логично и ясно описывают алгоритм.
- Вместе с тем, надо иметь в виду, что к вложению структур надо относиться очень внимательно. Главное, что надо помнить: это порядок выполнения действий, указанных на нагруженных дугах Р-схемы[10].

Задание 1. Читателю предлагается рассмотреть две Р-схемы. После просмотра определить порядок выполнения структур[11]:

10 Повторим короткую «формулу» этого правила: «Справа – налево, сверху – вниз».
[11] Сверить свой ответ с правильным можно, заглянув в раздел «Ответы к заданиям»

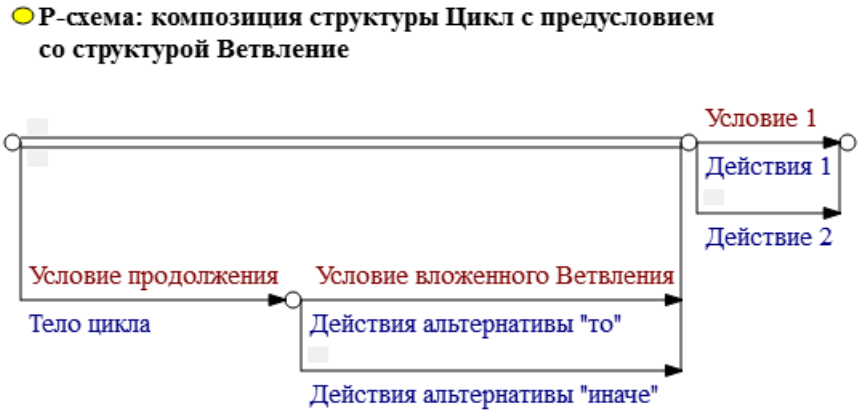


Рис. 4. 21 Р-схема № 1 задания 1

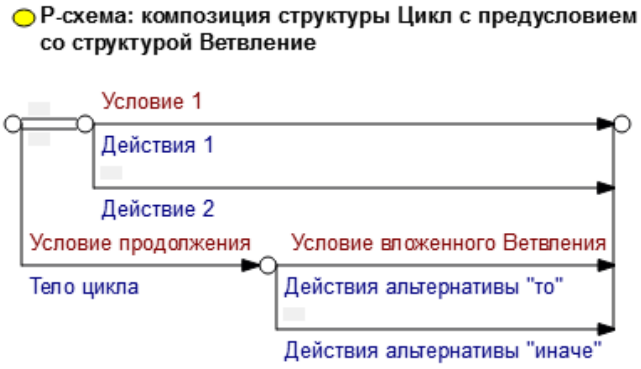


*Рис. 4. 22 Р-схема № 2 задания 1**<Назад>*

- Высокий уровень изобразительности и гибкости Р-схем демонстрирует следующий пример:

Рис. 4. 23 Иллюстрация разнообразия множественности вложений структуры Цикл

- Для просмотра исключительно структурной логики вложений структур алгоритма, *REditor* предоставляет возможность использовать режим просмотра «Скрыть текст».
- В этом случае мы получаем так называемую Р*-схему вида:

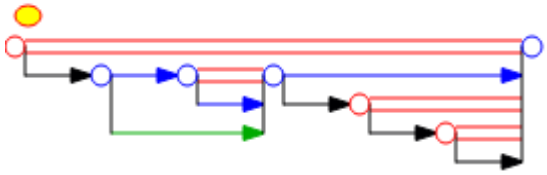


Рис. 4. 24 Р-схема аналог (логический «скелет») Р-схемы рис. 4.23*

- Опираясь на такую схему легко видеть, что
  - представленный ею алгоритм содержит четыре структуры *Цикл*;
  - тело внешней структуры *Цикл* состоит из *Следования* («Тело цикла 1») и двух последовательно расположенных структур *Ветвления 1* и *2*;
  - альтернатива «*то*» *Ветвления 1* состоит из *Следования* («*Действия альтернативы* «*то*»») и вложенного *Цикла 2*;
  - структура *Ветвление 1* полная, т.к. содержит альтернативу «*иначе*»;
  - структура *Ветвление 2* тоже полная, но ее альтернатива «то» – простая структура *Следование* («*Действия альтернативы* «*то*»»);
  - тело альтернативы «*иначе*» структуры *Ветвление 2* представляет собой «двукратный» *Цикл*, т.к. структура *Цикл 3* содержит в своем теле другую структуру цикл – *Цикл 4*.
- Оттранслированный вариант Р-схемы рисунка 4.23 имеет вид:

```
Р-схема : разнообразие вложенностей
управляющих структур
{
    while(Условие цикла 1){
        Тело цикла 1
        if(Условие ветвления 1){
            Действия альтернативы "то"
            while(Условие цикла 2){
                Тело цикла 2
            }
        }else{
            Действия альтернативы "иначе"
        }
        if(Условие ветвления 2){
            Действия альтернативы "то"
        }else{
            Действия альтернативы "иначе"
            while(Условие цикла 3){
                Тело цикла 3
                while(Условие цикла 4){
                    Тело цикла 4
                }
            }
        }
    }
}
```

Рис. 4. 25 Результат трансляции Р-схемы рисунка 4.23

- Представленный граф содержит 20 дуг, из которых 12 – горизонтальные и 4 из них специальные (двойные линии – признаки циклов).
- Это значит, согласно отмеченному выше, что для построения схем, изображенных на рис. 4.23, 4.24, после вызова элементарной Р-схемы потребуется выполнить
  - 4 двукратных нажатий левой клавиши мыши для получения специальных дуг;
  - 4 задержанных нажатий той же клавиши при соединении вершин (построение альтернативных ветвей);
  - 5 однократных нажатий той же клавиши для образования внутренних вершин.
- Отмеченное подчеркивает предельную экономичность механических действий разработчика алгоритма, благодаря чему основное внимание он может уделить обдумыванию логики решения задачи –главному элементу его работы.

К оглавлению К следующему параграфу К предыдущему параграфу

§5. КЛАССИФИКАЦИЯ ПРОСТЫХ ЦИКЛИЧЕСКИХ ПРОЦЕССОВ[12]

- Сознавая, что всякая классификация грешит условностью, не будем отрицать и того, что она (классификация) даёт реальные, практические инструменты более чёткого и точного выделения интересующего явления или процесса из всего множества исследуемых явлений и процессов.
- Не является в этом плане исключением и попытка классификации множества циклических процессов, которые составляют абсолютное большинство всех задач, решаемых на компьютерах.
- На первом этапе классификации отделим простейшие случаи от более сложных случаев. В связи с этим получим следующую схему:

Рис. 5. 1 Классификация циклических процессов

- Ко множеству *простых циклов* будем относить те циклические процессы, тело которых образуют только *простые команды* (операторы, структуры *Следование*).
- Названием «*составной цикл*» мы подчеркиваем то обстоятельство, что в состав *тела цикла* включены составные команды.
- К этой же группе мы отнесём и циклические алгоритмы, реализация которых требует привлечения сложных (составных) схем рассуждений[13].
- Продолжая дальнейшую классификацию простых циклов, будем придерживаться не традиционной классификации, принятой в программистской и методической литературе, но испытанной нами в ходе многолетней практики.
- В связи с этим выделим:

[12] Ознакомиться с иллюстрирующими Р-схемами данного параграфа можно в документе 02, папки 03-04, «03 Классификация циклов» и «04 Простые циклы» <Док. 02.03-04>

[13] В литературе помимо названия «*составные циклы*» используются такие термины, как «*сложные циклы*», «*вложенные циклы*», «*кратные циклы*».

Рис. 5. 2 Классификация простых циклических процессов

5.1 Арифметические циклы

- К этой группе простых циклов относятся циклические процессы *с заранее известным числом повторений тела* цикла. Таким образом, в этих процессах ещё до начала составления алгоритма программист с точностью до постоянной может указать количество повторений рабочей части.
- При реализации этих циклов всегда можно использовать *команду (оператор) цикла с параметром*.
- В циклах этого вида всегда присутствует величина, которую мы будем именовать *счетчик*. Эта величина по ходу реализации цикла встречается, по крайней мере, три раза:
  - на этапе *восстановления*. В это время счетчику присваивается некоторое исходное значение;
  - на этапе *контроля*. В ходе этого этапа значение счетчика сравнивается со значением эталона. *Эталон* – это специальная величина, задающая предельное значение счётчика;
  - на этапе *изменения*. В ходе реализации этого этапа значение счётчика изменяется на заданную величину.
- В А-циклах нередко имеется возможность построения *рекуррентных формул*.
- Заметим, что более лаконично приведенное словесное описание признаков, по которым простой цикл будет относиться к группе *А-циклов*, можно выразить посредством такой *Р-схемы*:

Рис. 5. 3 Характерные признаки А-циклов

<u>Определение</u> *Рекуррентной формулой* называется формула, выражающая некоторый член последовательности на основе одного или нескольких предыдущих её членов и, возможно, номера строящегося члена.

- Формальными рекомендациями к применению рекуррентных формул служат следующие признаки:
  - член последовательности или его часть содержит в своём составе степень с постоянным основанием;
  - член последовательности или его часть содержит в своём составе факториал.
- Заметим, что приведённые признаки являются *необходимыми* показаниями к использованию рекуррентных формул.
- Практика программирования показывает, что помимо этого есть случаи, «не вписывающиеся» в приведённую схему. В этих случаях рекуррентные формулы строятся индивидуально для каждой конкретной ситуации.
- Рассмотрим конкретный пример построения алгоритма на А-циклы с возможностью привлечения рекуррентной формулы и схему общих рассуждений в ходе её построения.

Задача 1. Найти значение суммы $\sum_{k=0}^{n} \frac{(-1)^{k+1} x^{2k-1}}{(2k-1)(2k+1)!}$.

Спецификация ввода: ***значения n, x.***

Спецификация вывода: ***значение суммы.***

Решение

- Во-первых, обратим внимание на то, что слагаемое предлагаемой суммы допускает частичное использование рекуррентной формулы, так как в знаменателе есть сомножитель (***2k–1)***, который «не вписывается» в рекомендации использования рекуррентных формул.

- Заметим, что включение этого сомножителя в формулу не сделает ее неправильной (при корректных выкладках), но не приведет лишь к сокращению вычислений!

Рис. 5. 4 Порядок решения задачи

- Практика показывает, что наиболее ответственным и «уязвимым» на ошибки является первый этап «*Построение рекуррентной формулы*», поэтому приводим алгоритм его выполнения:

Построение рекуррентной формулы. Цель: выразить A[n+1] через A[n]

Выделить ту часть формулы, которая содержит
- операции возведения в степень (основание постоянно)
- просчет факториала

Выделенную часть записать в виде функции от параметра k :

$$a_k = \frac{(-1)^{k+1} x^{2k-1}}{(2k+1)!}$$

Записать формулу на основе построенной, но со значением параметра k+1 :

$$a_{k+1} = \frac{(-1)^{k+2} x^{2k+1}}{(2k+3)!}$$

Разделить последующий член на предыдущий и выполнить сокращение :

$$\frac{a_{k+1}}{a_k} = \frac{(-1)^{k+2} x^{2k+1} (2k+1)!}{(2k+3)!\,(-1)^{k+1} x^{2k-1}} = \frac{-x^2}{(2k+2)(2k+3)}$$

Умножить обе части последнего выражения на a[k] :

$$a_{k+1} = \frac{-a_k x^2}{(2k+2)(2k+3)}$$

На основе формулы для a[k], просчитать значение первого члена рекуррентной формулы :

$$a_0 = \frac{(-1)^{0+1} x^{2*0-1}}{(2*0+1)!} = -\frac{1}{x}$$

Результат :

$$a_{k+1} = \frac{-a_k x^2}{(2k+2)(2k+3)}, \qquad a_0 = -\frac{1}{x}$$

Рис. 5. 5 Алгоритм построения рекуррентной формулы

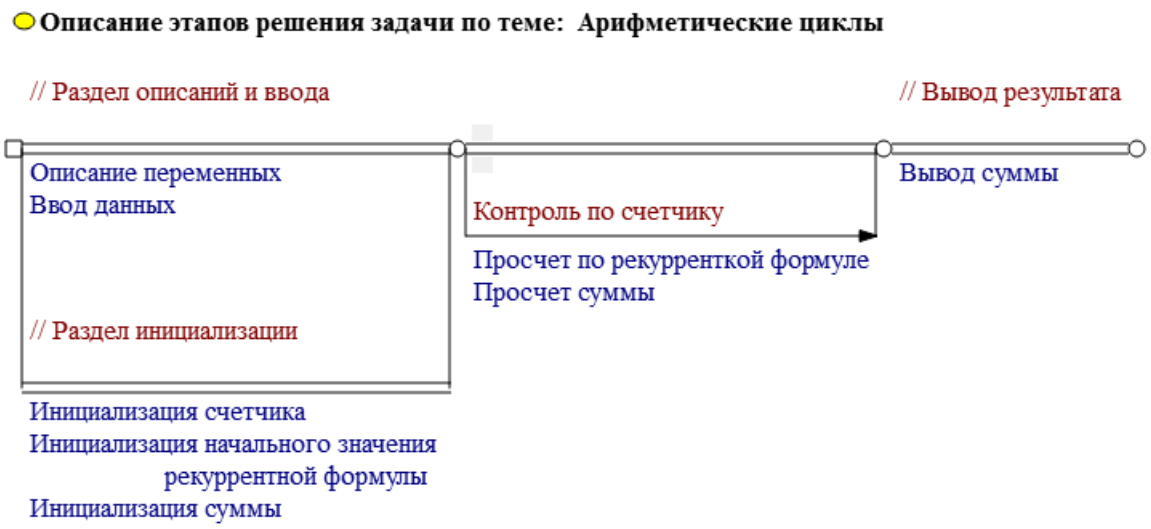


Рис. 5. 6 Корневой уровень детализации

- Заметим, что представленный на рис. 5.6 алгоритм записан в самом общем достаточно детализированном виде. Далее можно его уточнять под любой язык программирования.
- Первой реализацией предлагается школьный алгоритмический язык *МегаЕ*, который можно, кстати, использовать как отечественный псевдокод[14].

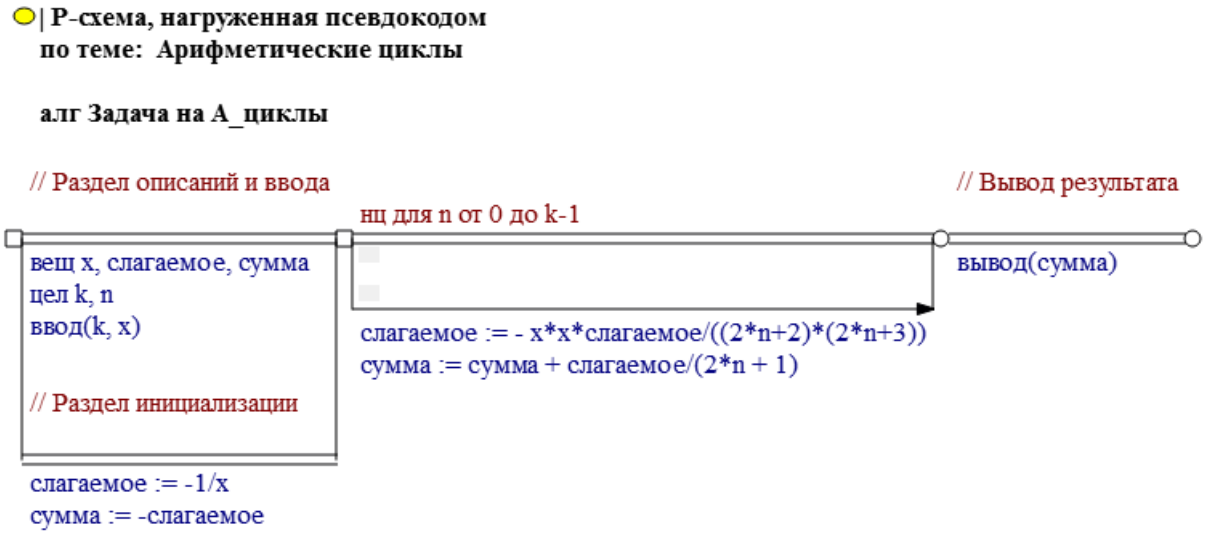


Рис. 5. 7 Детализация алгоритма средствами языка МегаЕ

- Самым замечательным моментом подобного представления является то, что исходный код, получаемый в результате компиляции средствами *REditor*, оказывается практически таким же, каким получается в среде разработки *МегаЕ*-кода *КУМир*.

[14] Обратим внимание, что в подавляющем количестве переводной литературы, посвященной разработке алгоритмов, а не описанию языка, авторы чаще всего избирают свою (авторскую и естественно, англоязычную) нотацию записи, а не средства Pascal, C или иные. См., например, [7 – 10].

- Аналогичная ситуация наблюдается и с другими языками программирования:

```
program ACycleTask_01;
{   Напишите программу нахождения суммы, включая в нее первые k слагаемых:
                                3                          k+1   2k-1
          1          x        x                         (-1)    * x
     S = -------- + -------- - -------  + . . . + -------------------------
         x*1!       1*3!     3*5!                    (2k - 1)*(2k + 1)!

    Спецификация ввода :
       k x
    Спецификация вывода:
       S
}

{$APPTYPE CONSOLE}

uses
 SysUtils;

 var
  x, summa, slagaemoe : double;
  k, n : integer;
```

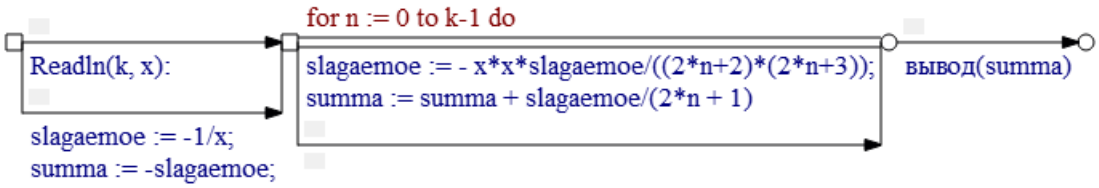


Рис. 5. 8 Детализация алгоритма средствами языка Delphi[15]

15 Справедливости ради надо заметить, что использованная среда *REdit* ориентирована на язык C++, поэтому скомпилированный ею код необходимо корректировать.

```
using System;

namespace АциклЗадача_01;
/*   Напишите программу нахождения суммы, включая в нее первые k слагаемых:
*                                 3                        k+1    2k-1
*            1            x      x                      (-1)     * x
*      S =  -------- + -------- - -------  + . . . + ------------------------
*           x*1!       1*3!    3*5!                 (2k - 1)*(2k + 1)!
*
*    Спецификация ввода :
*       k x
*    Спецификация вывода:
*       S
*/
     class Program
```

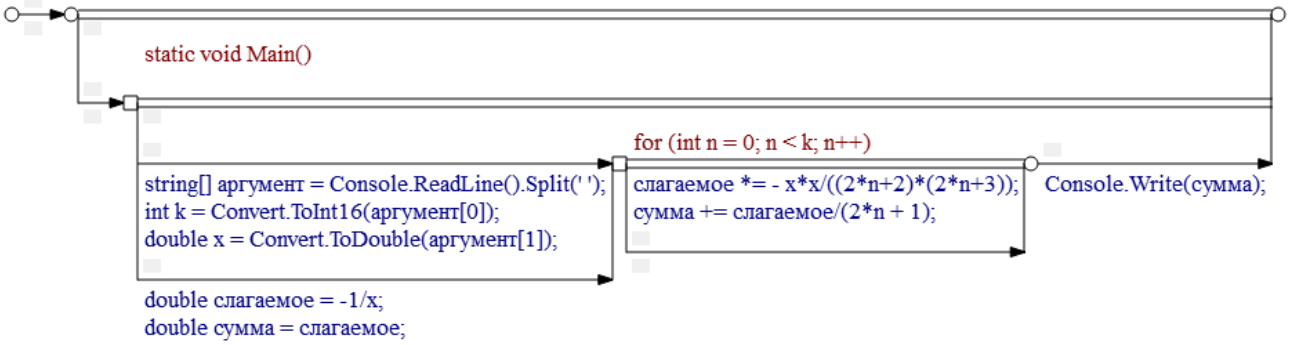


Рис. 5. 9 Детализация алгоритма средствами языка C#[16]

```
// Tsk_01_A_cycle_Cpp.cpp: определяет точку входа для консольного приложения.

/*   Напишите программу нахождения суммы, включая в нее первые k слагаемых:
                                 3                        k+1    2k-1
            1            x      x                      (-1)     * x
      S =  -------- + -------- - -------  + . . . + ------------------------
           x*1!       1*3!    3*5!                 (2k - 1)*(2k + 1)!

    Спецификация ввода :
       k x
    Спецификация вывода:
       S
*/
#include "stdafx.h"
#include <iostream>
#include <math.h>

void _tmain(int argc, _TCHAR* argv[])
```

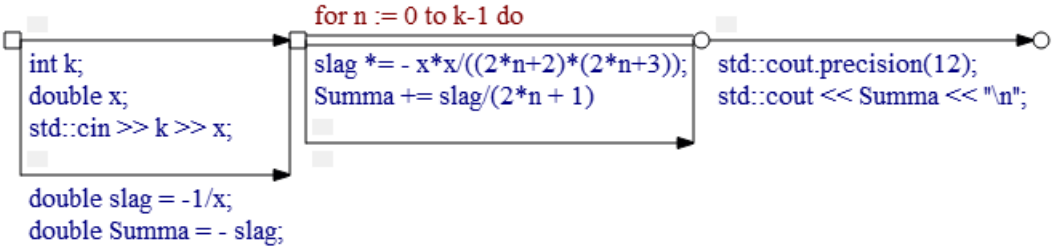


Рис. 5. 10 Детализация алгоритма средствами языка C++ (консольное приложение Win32)

[16] Заметим, что язык программирования C# изначально объектно-ориентированный. Поэтому при построении даже элементарных программ используется блок class и указывается пространство имен (namespace). В приведенном примере мы преднамеренно указанные строки не указали, сконцентрировав внимание только на оформлении основной функции Main.

```
#-*- coding: utf -8 -*-

namespace АциклЗадача_01;
"""  Напишите программу нахождения суммы, включая в нее первые k слагаемых:
                                3                    k+1    2k-1
          1          x      x                     (-1)     * x
     S = -------- + -------- - -------  + . . . + ----------------------
         x*1!       1*3!    3*5!                 (2k - 1)*(2k + 1)!

    Спецификация ввода :
       k x
    Спецификация вывода:
       S
"""
```

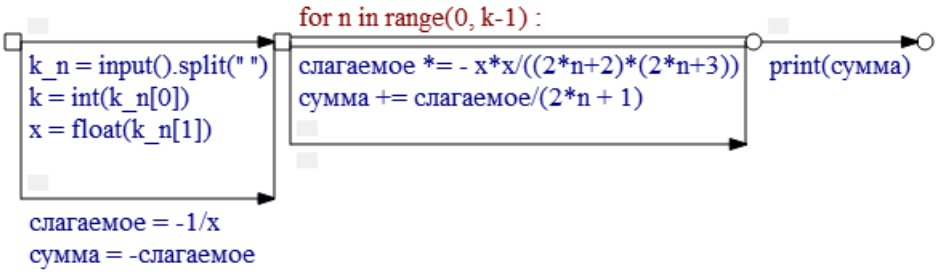


Рис. 5. 11 Детализация алгоритма средствами языка Python

Вывод 1: *Сравнивая корневой (рис. 5.7) и детализированные алгоритмы рис. 5.8 – 5.11 , можно заключить, что*

- корневой алгоритм в полной мере определяет узловые моменты детализированных алгоритмов;
- детализированные алгоритмы, описанные средствами высокоуровневых языков, отличаются друг от друга незначительными деталями. В данном случае – порядком описания типов переменных, особенностями написания символа присваивания, но более всего – процедурами ввода-вывода.

- *Следовательно,* при изучении основ программирования главное внимание должно быть уделено построению алгортима. В этом отношении средства двумерного программирования *Р-схем* подходят наилучшим образом.
- Кроме того, при рассмотрении различных языков программирования особое внимание надо будет обратить на средства ввода-вывода.

5.2 Итерационные циклы[17]

- Опираясь на опыт описания предыдущего пункта, в данном пункте ограничимся достаточно информативным описанием сути И-циклов с помощью Р-схем.

Итерационные циклы

Циклические процессы, реализующие итерационные методы вычислительной математики

Этот вид процессов характерен неопределенным заранее числом повторения рабочей части (тела) цикла

Организиция контроля за окончанием циклический повторений:
- по малости невязки;
- по близости соседних приближений

В этом виде процессов практически всегда используюся рекуррентные формулы, определяемые вычислительным процессом

Рис. 5. 12 Характерные признаки И-циклов

Виды простейших итерационных методов

Метод Ньютона :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Модифицированные метод Ньютона :

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_0)}$$

Метод секущих :

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - c)}{f(x_n) - f(c)}$$

Метод половинного деления , определяется шагами вычислений

Рис. 5. 13 Рекуррентные формулы И-циклов

[17] Подробное рассмотрение теории итерационных методов будет осуществлено в курсе «*Численные методы*». Здесь же ограничимся общими пояснениями их сути. Пусть дано уравнение $f(x) = 0$. На основе некоторых соображений (они определяются идеями построения используемого итерационного метода) строится новое уравнение вида $x = \varphi(x)$, эквивалентное исходному уравнению, по крайней мере, в некоторой окрестности отыскиваемого решения. Вид преобразованного уравнения называется видом удобным для итерации. Именно это уравнение порождает рекуррентную (итерационную) формулу для расчётов: $x_{k+1} = \varphi(x_k)$ где x_0 – начальное приближение (точка, взятая из области сходимости метода). На основе последовательного применения рекуррентной (итерационной) формулы получается последовательность приближённых решений $x_0, x_1, x_2, \ldots x_n, \ldots \to x^*$, где x* – решение исходного уравнения.

Рис. 5. 14 Детализация шага итерации метода половинного деления

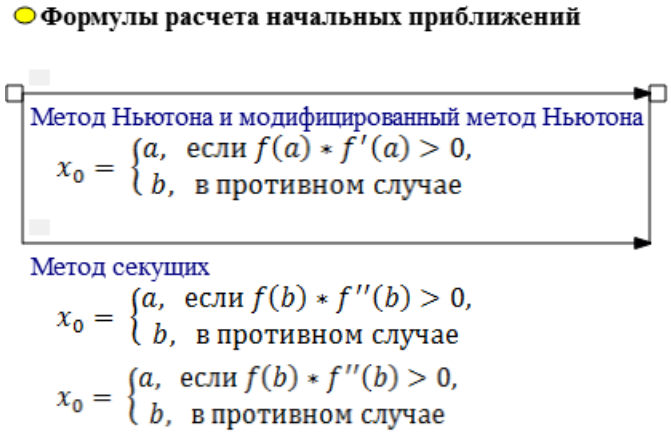


Рис. 5. 15 Формы расчета начальных приближений И-циклов

- В данном случае ***a*** и ***b*** – концы отрезка, содержащего корень уравнения. Для метода половинного деления формула не указана, т.к. согласно *Р-схеме шага ме*тода (рис. 5.14), в нем задействованы оба конца.

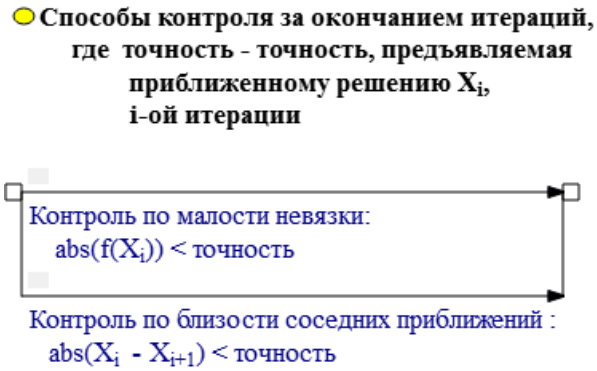


Рис. 5. 16 Организация контроля в И-циклах

Задача 2 Решить уравнение $f(x) = x + \ln(x + 0.5) - 0.5 = 0$ методом Ньютона с контролем за окончанием просчётов по малости невязки, если известно, что отыскиваемый корень этого уравнения находится на промежутке [0, 2].

Спецификация ввода: *a, b, точность (где a = 0, b = 2).*
Спецификация вывода: значение решения.
Решение.

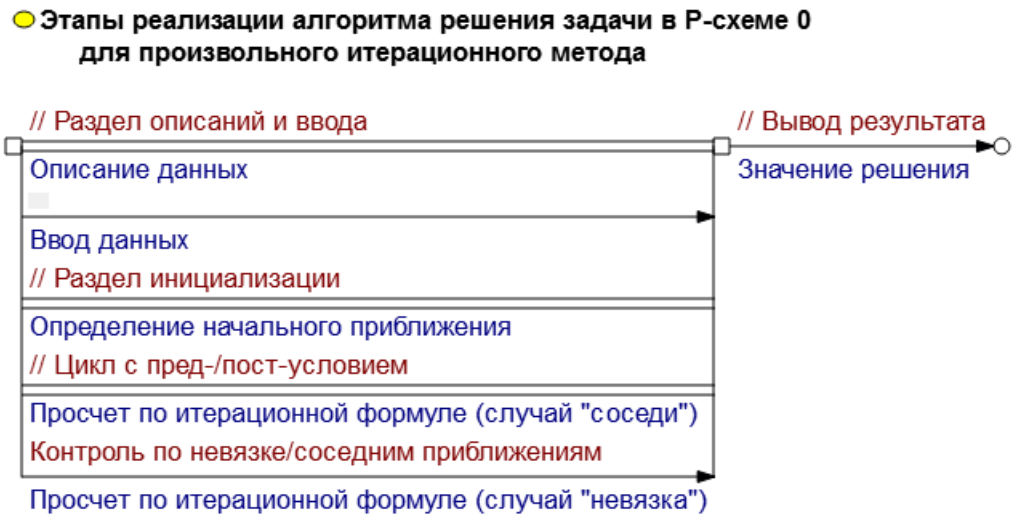


Рис. 5. 17 Общая схема реализации И-циклов[18]

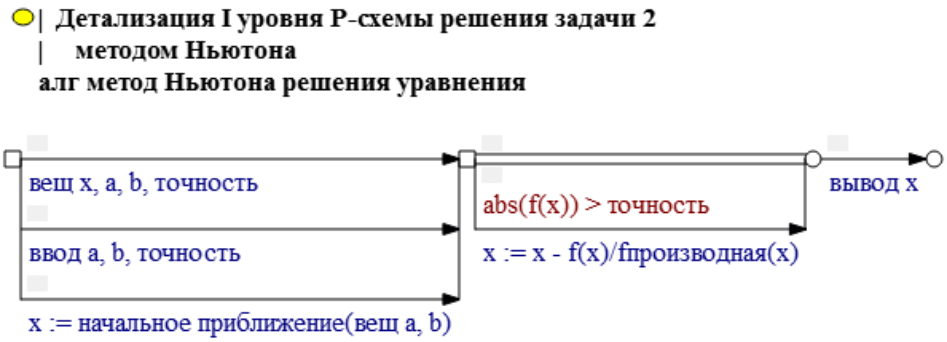


Рис. 5. 18 Реализация алгоритма метода Ньютона в обозначениях языка МегаЕ

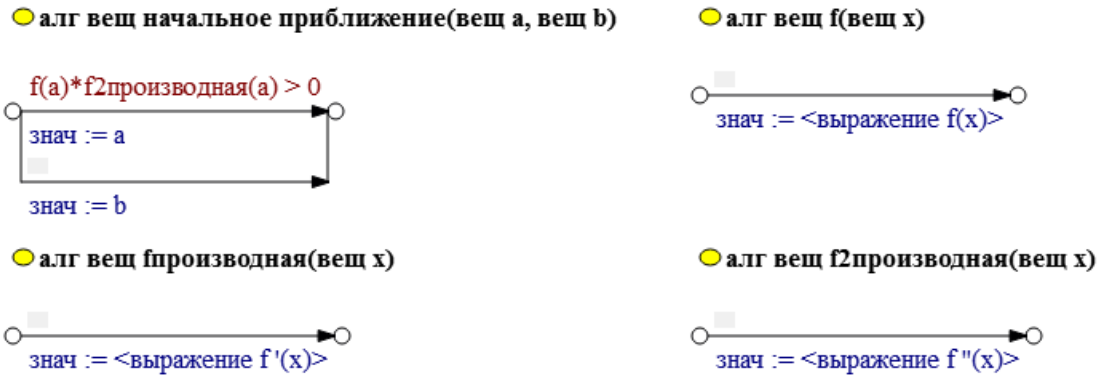


Рис. 5. 19 Вспомогательные функции для головного алгоритма метода Ньютона

- И, наконец, детализируем алгоритм в «традициях» языка программирования Delphi:

[18] Этап алгоритма «Цикл с пред-/пост-условием» должен содержать только один вариант реализации тела цикла «Просчет по итерационной формуле …». В случае «Цикл с предусловием» – использовать «Контроль по невязке» и «Просчет по итерационной формуле (случай «невязка»)», а в «Цикл с постусловием» – использовать «Контроль по соседним приближениям» и «Просчет по итерационной формуле (случай «соседи»)».

- во-первых, Delphi-код должен начинаться заголовком и подключением библиотек, после чего расположим все вспомогательны алгоритмы;

```
program ICycleTask_01;
{   Решить уравнение:

        x  + ln(x + 0.5) - 0.5 = 0;

  используя метод Ньютона:
                          f(x_n)
        x_(n+1) = x_n - -----------
                          f'(x_n)
  Корень уравнения находится на отрезке [0; 2]. Контроль
  за окончанием просчетов проводить по малости невязки.

    Спецификация ввода:
        a, b, точность.
    Спецификация вывода:
        значение решения.
}

{$APPTYPE CONSOLE}

uses
  SysUtils;

Function f(x : double) : double;

Result := x + ln(x + 0.5) - 0.5

  Function fDeriv(x : double) : double;

  Result := 1 + 1/(x + 0.5)

  Function fSDeriv(x : double) : double;

  Result := - 1/sqr(x + 0.5)

  Function BeginApproximation(a, b : double) : double;

  f(a)*fDeriv(a) > 0
  Result := a
  Result := b
```

Рис. 5. 20 Раздел описания вспомогательных алгоритмов

- во-вторых, опишем глобальные переменные и головной алгоритм.

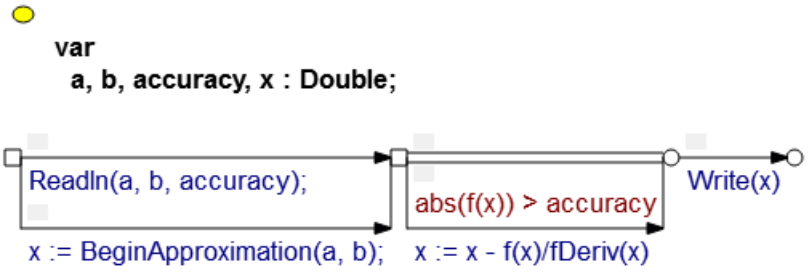


Рис. 5. 21 Организация контроля в И-циклах

- В результате редактор *REditor* генерирует замечательный код (правда, в стиле C++), который требует лишь незначительной корректировки:

```
program ICycleTask_01;
{    Решить уравнение:

        x  + ln(x + 0.5) - 0.5 = 0;

   используя метод Ньютона:
                          f(x_n)
         x_(n+1) = x_n - -----------
                          f'(x_n)
   Корень уравнения находится на отрезке [0; 2]. Контроль
   за окончанием просчетов проводить по малости невязки.

     Спецификация ввода:
         a, b, точность.
     Спецификация вывода:
         значение решения.
}

{$APPTYPE CONSOLE}

uses
  SysUtils;

Function f(x : double) : double;
{
      Result := x + ln(x + 0.5) - 0.5
}
```

```
Function fDeriv(x : double) : double;
{
    Result := 1 + 1/(x + 0.5)
}

Function fSDeriv(x : double) : double;
{
    Result := - 1/sqr(x + 0.5)
}

Function BeginApproximation(a, b : double) : double;
{
    if(f(a)*fDeriv(a) > 0){
        Result := a
    }else{
        Result := b
    }
}

 var
   a, b, accuracy, x : Double;
{
    Readln(a, b, accuracy);
    x := BeginApproximation(a, b);
    while(abs(f(x)) > accuracy){
        x := x - f(x)/fDeriv(x)
    }
    Write(x)
}
```

Рис. 5. 22 Исходный код, сгенерированный компилятором редактора REditor

- Заметим в заключение, что недостаточно аккуратная математическая подготовка к решению задачи на И-циклы чревата «плачевными последствиями». А именно, алгоритм и программа могут быть с точки зрения информатики составлены безупречно, но решение задачи не будет получено. Под математической подготовкой задачи понимается:
  - аккуратное нахождение всех производных;
  - аккуратное определение промежутка содержания начального приближения (если его надо найти самостоятельно);
  - точное определение длины промежутка, гарантирующее сходимость используемого итерационного метода. Это замечание особенно важно для метода Ньютона;
  - аккуратное определение начального приближения.

5.3 Циклы с Контролем за Монотонной Величиной

Циклы с Контролем за Монотонной Величиной

В задачах этого типа присутствует хотя бы одна монотонно изменяющаяся величина

В этом виде циклических процессов заранее не известно, сколько раз будет повторена по ходу вычислений рабочая часть цикла (тело цикла)

В этом виде циклических процессов нередко используется (по аналогии с А-циклами) Счетчик

Большая возможность использования рекуррентных формул

Контроль за окончанием цикла ведется за счет отслеживания (контроля) за возрастанием/убыванием Монотонной Величины (см. п. 1)

Рис. 5. 23 Характерные признаки КМВ-циклов

Счетчик

Этап : Восстановление
Счетчик <== начальное значение

Этап : Модификация
Счетчик <== Счетчик +/- Шаг

Рис. 5. 24 Особенности Счетчика в КМВ-циклах

- Обратим внимание на то, что рассматриваемый вид простых циклов имеет общие черты с двумя предыдущими. И вместе с тем – это особый вид простых циклов. Несмотря на то, что в *КМВ-циклах*, как и в *А-циклах*, нередко используется *счетчик*, он в этом случае не используется для контроля за окончанием. Его единственное назначение – поддержка рекурретных формул. Неопределенность количества повторений тела *КМВ-цикла*, «роднящего» его с *И-циклом*, вызвана стремлением повысить экономичность вычислений (по сравнению с *А-циклами*), выполняя их лишь до тех пор, пока не достигнута требуемая точность (или при возрастании подконтрольной монотонной величины, пока последняя впервые не превысит допустимый рубеж).

Задача 3 Вычислить сумму: $\sum_{k=0}^{\infty} \frac{x^{2k}}{2^k * k!}$, продолжая суммирование до тех пор, пока значения слагаемых велики по отношению к заданной величине ε (точность).

Спецификация ввода: *x, точность.*
Спецификация вывода: *значение суммы.*
Решение.

- Предварительный порядок рассуждений в данном случае точно такой же, как и для А-цикла, и описан Р-схемой рис. 5.4. Более того, построение рекуррентной формулы тоже строится аналогично Р-схеме рис. 5.5. Тем не менее, мы вновь приведем аналогичные построения для данного примера:

Построение рекуррентной формулы.
Цель: выразить A[k+1] через A[k]

Выделить ту часть формулы, котрая содержит
- операции возведения в степень (при постоянном основании)
- просчет факториала

Выделенную часть записать в виде функции от параметра k :

$$a_k = \frac{x^{2k}}{2^k * k!}$$

Записать формулу на основе построенной, со значением параметра k + 1 :

$$a_{k+1} = \frac{x^{2k+2}}{2^{k+1} * (k+1)!}$$

Построить частное от деления последующего члена (A[k+1]) на предыдущий (A[k]) и выполнить сокращения :

$$\frac{a_{k+1}}{a_k} = \frac{x^{2k+2} * 2^k * k!}{2^{k+1} * (k+1)! * x^{2k}} = \frac{x^2}{2 * (k+1)}$$

Умножить обе часть полученного выражения на A[k] :

$$a_{k+1} = \frac{x^2 * a_k}{2 * (k+1)}$$

Используя формулу для A[k], найти начальный член последовательности, в данном случае это A[0] :

$$a_0 = 1$$

Объединяя результаты двух последних шагов, получаем рекуррентную формулу-результат :

$$a_{k+1} = \frac{x^2 * a_k}{2 * (k+1)}, \quad a_0 = 1$$

Рис. 5. 25 Алгоритм построения рекуррентной формулы для КМВ-цикла

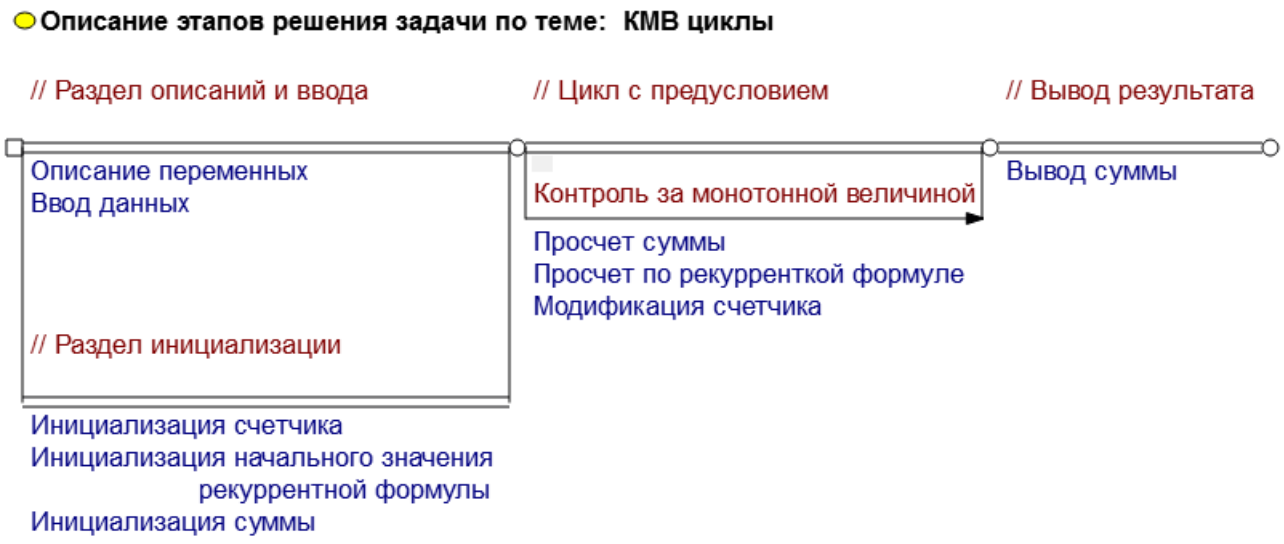


Рис. 5. 26 Алгоритм этапов реализации КМВ-цикла

- Детализируем алгоритм в «традициях» языка программирования Delphi:

```
program CMVCycleTask_01;
{   Вычислить сумму вида:
                  2        4       6                 2k
                 x        x       x                 x
      S = 1 + ------- + ------- + -------   +  . . .  + ------------- + . . .
                2*1!     4*2!    8*3!                  2^k * k!

  продолжая суммирование до тех пор, пока значения слагаемых
  велики по отношению к заданной величине точность.

      Спецификация ввода :
          x точность
      Спецификация вывода:
          S
  }

  {$APPTYPE CONSOLE}

  uses
    SysUtils;

    var
      x, summa, slagaemoe, accuracy : double;
      k : integer;
```

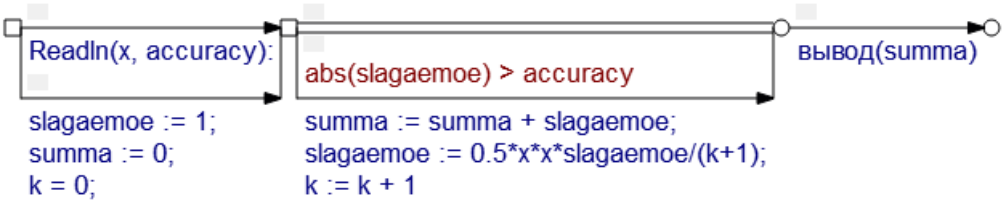


Рис. 5. 27 Детализированный головной алгоритм КМВ-цикла в обозначениях Delphi

***Вывод** 2: Используя Р-схемы общих алгоритмов реализации соотвественно А-, И- и КМВ-циклов (рис. 5.6, 5.17, 5.26) и построив на основе их «бестекстовые» варианты, получим алгоритмы Р*-алгоритмы вида:*

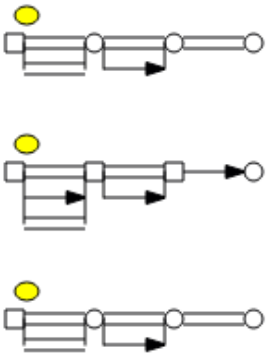


Рис. 5. 28 Р-схемы общих алгоритмов А-, И- и КМВ-циклов*

- во-первых, схемы рис. 5.28 проявляют скелеты алгоритмов, четко показывают компоновку управляющих структур алгоритмов;
- во-вторых, подтверждают «законность» объединения рассмотренных видов циклов в одну группу «Простых циклов» с теми их особенностями, о которых говорилось в начале §5.

***Следовательно,** средства Р-схем помогают быстрее, проще и четче устанавливать общие особенности внешне отличных друг от друга алгоритмов.*

5.4 Циклы с Внутренним Контролем

- К этому типу задач мы будем относить задачи, которые «не вписываются» ни в одну из ранее описанных схем определения вида простого цикла. Это процессы, о которых можно сказать с определённостью только одно – они циклические, причем они напоминают бесконечные циклы.
- С другой стороны, использование схем А-, И- и КМВ-циклов в «чистом виде», иногда может привести к «зацикливанию»,
  - как в смысле получения без конца повторяющихся вычислений,
  - так и в смысле многократно «впустую» выполняемых операций, фактически не влияющих на конечный целевой результат вычислений.

    В первом случае, такие явления могут наблюдаться при реализации И- и КМВ-циклов тогда, когда точность требуемых вычислений (контрольный эталон) завышена. Во втором случае, это может наблюдаться в случае построения А-цикла, опять же с завышенным значением количества повторений тела цикла.

    Оба случая – результат того, что при обработке данных вещественного типа компьютерные вычисления выполняются с заведомо ограниченной

(разрядной сеткой компьютера) точностью и с накоплением, в связи с этим, погрешностью округления.

- Для контроля окончания просчётов
  - в первом случае вводят некоторую искусственную величину того же типа, что и величины ввода, но с таким значением, которое либо не может встретиться в потоке ввода, либо встреча такого значения очень маловероятна;
  - во втором случае – вводят дополнительный контроль над значением величины, которая может быть источником «пустых» действий или вызвать зацикливание.
- Ограничимся в данном случае не столько деталями возможных примеров, сколько схемой разрешения конфликта.
- Итоговые характеристики, сведенные в Р-схему, таковы:

Циклы с дополнительным Внутренним Контролем

Случай 1 : Циклический процесс "не вписывается" ни в одну из выше приведенных схем (А-, И- и КМВ-циклы)

Случай 2 : Арифметический цикл с "заведомо" большим значением эталона

Случай 3 : Итерационный цикл с "заведомо" завышенной точностью просчетов

Случай 4 : КМВ - цикл с "заведомо" большой/малой контрольной величиной эталона

Для случая 1 : вводится специально подобранная величина, достижение значения которой - сигнал для выхода из цикла

Для случаев 2-4 : вводится дополнительный контроль за величиной, поведение которой может служить источником
- либо "лишних" просчетов;
- либо "зацикливания".

Рис. 5. 29 Основные характеристики алгоритмов ВнК-цикла

Задача 4 Организовать ручной ввод и суммирование потока целочисленных значений, о котором известно, что все его значения неотрицательны.

***Спецификация ввода**: последовательность чисел (закончить числом –1).*
***Спецификация** вывода**:** значение суммы.*

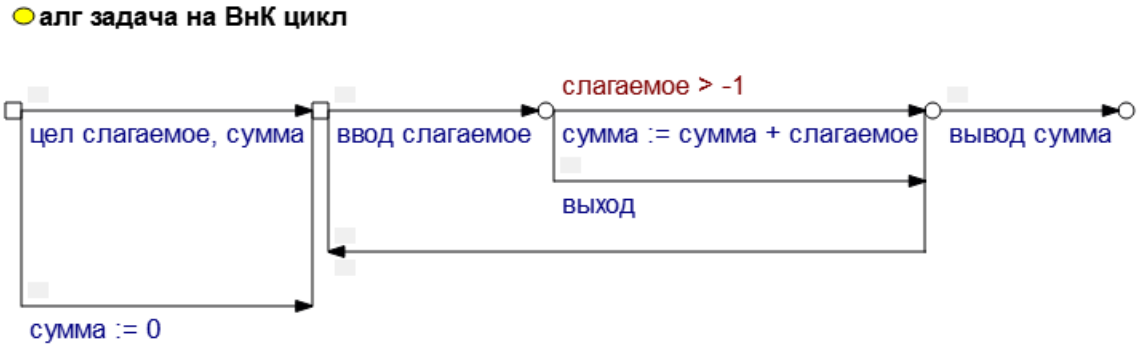


Рис. 5. 30 Детализированный головной алгоритм ВнК-цикла в обозначениях МегаЕ

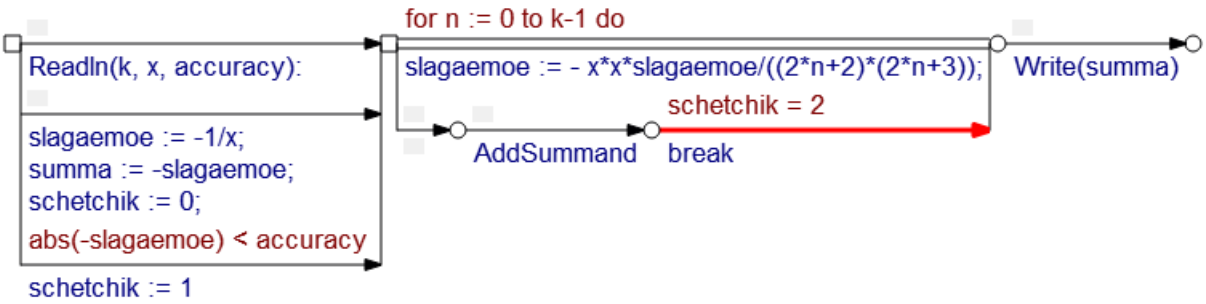
- В данном примере в качестве контрольной величины выступает константа ***–1***. Выбор константы в качестве контрольного значения окончания *ВнК – цикла* следует признать удовлетворительным, так как по условию задачи сказано, что все *слагаемые неотрицательные*.

```
program A_InContr_CycleTask_01;
{   Напишите программу нахождения суммы включением в нее первых k слагаемых:
                                  3                    k+1    2k-1
                 1        x       x                   (-1)     * x
        S =   ------- + ------- - -------   +  . . . + ------------------------
              x*1!     1*3!    3*5!                  (2k - 1)*(2k + 1)!

до тех пор, пока два последовательно расположенных слагаемых по модулю были
больше контрольной величины ТОЧНОСТЬ.
В этом случае предпоследнее слагаемое в сумму включать должно, а последнее - нет.
     Спецификация ввода :
        k x точность
     Спецификация вывода:
        S
}
{$APPTYPE CONSOLE}

uses
  SysUtils;

  var
    x, summa, slagaemoe, accuracy : double;
    k, n, schetchik : integer;
```




```
Procedure AddSummand;
```

Рис. 5. 31 Детализированный головной алгоритм А-ВнК-цикла в обозначениях Delphi

- В данном примере в отличие от *Р-схемы*, приведенной на рис. 5.8, в качестве контрольной величины выступает дополнительная величина ***точность***. Выбор этой величины в качестве контрольного значения за окончанием *А-ВнК – цикла* следует признать удовлетворительным, так как она «не дает», по аналогии с КМВ-циклом, включать в сумму «машинные нули».
- *Р-схемы* простых циклов с дополнительным внутренним контролем общего вида могут иметь такой вид:

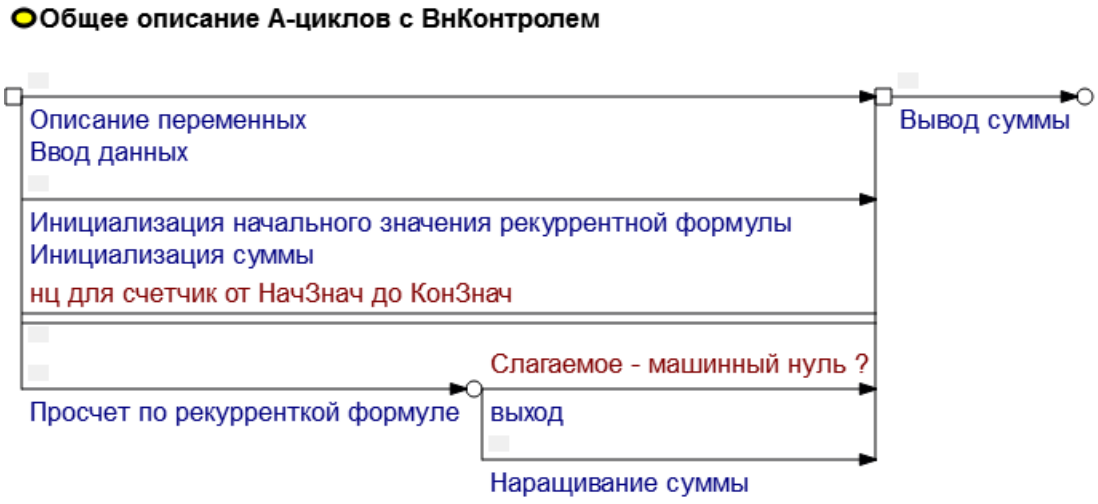


Рис. 5. 32 Общая Р-схема А-ВнК-цикла

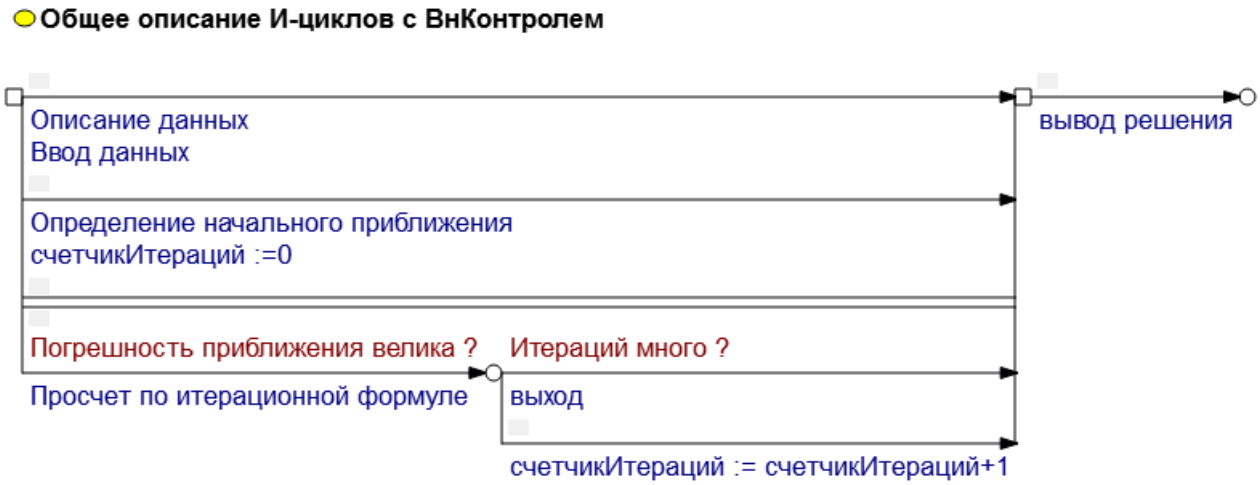


Рис. 5. 33 Общая Р-схема И-ВнК-цикла

Рис. 5. 34 Общая Р-схема КМВ-ВнК-цикла

К оглавлению К следующему параграфу К предыдущему параграфу

§6. КЛАССИФИКАЦИЯ ЦИКЛИЧЕСКИХ ПРОЦЕССОВ ОБРАБОТКИ МАССИВОВ[19]

- Для введения понятия «таблица» (массив), рассмотрим простейшую задачу:

> ***Задача*** 5 Составить алгоритм вычисления периметра треугольника, зная длины всех его сторон : a, b, c.

- Легко записывается алгоритм:

```
| Задача 5
|   Составить алгоритм вычисления периметра треугольника,
| зная длины всех его сторон : a, b, c.

алг вещ периметр(арг вещ a, b, c)
```

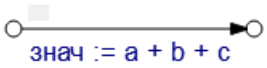


Рис. 6. 1 Р-схема функции просчета периметра треугольника

- Обобщая задачу, получаем:

> ***Задача*** 6 Составить алгоритм вычисления периметра 5-ти, 100-угольника, зная длины всех его сторон: a_1, a_2, ... , a_{100}.

- «Копируя» предыдущее решение, получаем:

```
| Задача 6
|   Составить алгоритм вычисления периметра 5-ти, 100-угольника,
| зная длины всех его сторон : a1, a2, ... , a100 .

алг вещ периметр (арг вещ a1, a2, ... , a100)
```

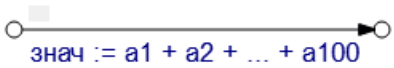


Рис. 6. 2 Р-схема функции просчета периметра 100-угольника

- Однако приведенное решение неприемлемо, по крайней мере по двум причинам:
  - в записи выражений на любом языке программирования ***недопустимо*** использовать *подстрочные индексы*;
  - в записи выражений на любом языке программирования ***недопустимо*** использовать *многоточие, заменяющее члены последовательностей*.

[19] Ознакомиться с иллюстрирующими Р-схемами данного параграфа можно в документе 03, папки 01, «01 Обработка таблиц обзор» и 02 – «02 Образцы решения задач» <Док. 03.01-02>

- Отмеченные причины фактически акцентировали внимание на *основных свойствах* ***табличных величин***:
  - таблица состоит из множества простых переменных;
  - – к элементам таблицы можно обратиться *не по уникальным именам*, а с указанием имени таблицы и *номера элемента (индекса)*.
- Следующий вопрос, на который следует отвечать при решении реальных задач: *когда надо* и *когда не следует* объединять простые переменные в ***табличные величины***.
- Ответ на поставленный вопрос содержится в следующей характеристике табличной величины:

  Табличная величина (массив) – это *составная* (структурная, многокомпонентная) величина, хранящая *однородную* информацию.

  Однородность означает, *во-первых, равноправность* элементов/компонент (все компоненты имеют одинаковый содержательный смысл, над ними выполняются одинаковые операции); *во-вторых, независимость от количества.* Таким образом, если информация однородна, то знание точного количества элементов табличной величины не влияет на работу составленного для ее обработки алгоритма.
- Далее, объединяя задачи по *общности условий* и *общности решений*, выделим вслед за Я.Н. Зайдельманом[20] следующие группы задач:
  - – Задачи на *заполнение* линейных табличных величин (векторов);
  - – Задачи на *анализ* содержимого линейных таблиц;
  - – Задачи на *поиск* информации в линейных таблицах;
  - – Задачи на *преобразование* (сортировку) информации в линейных таблицах.

Рис. 6. 3 Классы задач по обработке табличных (массивовых, списочных) величин

[20] Зайдельман Я.Н., Табличные величины, газ. «Первое сентября», еженед. прилож. «Информатика», №19, май 1996

6.1 Задачи на заполнение табличных величин

- В задачах этой группы табличная величина – это результат выполнения алгоритма. В силу сказанного, заголовок подобных алгоритмов в общем случае может выглядеть следующим образом:

```
| Образец построения заголовка подпрограммы
| на заполнение линейного массива
| в обозначениях МегаЕ

алг построение таблицы(арг цел n, рез цел таб a[1 : n])
```

Рис. 6. 4 Заголовок функции типовой задачи на заполнение (МегаЕ)

```
| Образец построения заголовка подпрограммы
| на заполнение линейного массива
| в обозначениях Delphi
|   где
|     type
|       vector = array of integer;

procedure buildVector(out a : vector);
```

Рис. 6. 5 Заголовок функции типовой задачи на заполнение (Delphi)

```
| Образец построения заголовка подпрограммы
| на заполнение линейного массива
| в обозначениях C#

void построениеТаблицы(out int[] a)
```

Рис. 6. 6 Заголовок функции типовой задачи на заполнение (C#)

- Заметим, что в конкретных ситуациях в заголовке могут появляться дополнительные формальные параметры.
- По методу решения задачи этой группы тоже могут быть подразделены на
  - задачи *прямого заполнения* табличных величин и
  - задачи *заполнения с поиском*.
- В первом случае каждый элемент строящейся таблицы *непосредственно вычисляется* (с привлечением рекуррентных формул, обычных формул или датчика случайных чисел) по ходу работы алгоритма.
- Примерами задач этого класса могут быть такие:
  - заполнить элементы таблицы нулями (заданным значением);
  - заполнить таблицу квадратами натуральных чисел;

- заполнить таблицу последовательностью четных/нечетных чисел;
- заполнить таблицу числами Фибоначчи ($F_0 = F_1 = 1$; $F_{i+1} = F_i + F_{i-1}$);
- заполнить таблицу значениями чисел, вводимыми с клавиатуры;
- заполнить таблицу случайными числами из интервала [***a, b***] и т.д.

- Во втором случае каждый элемент строящейся таблицы получается в результате операции *поиска*. В этом случае получение общей формулы элемента таблицы невозможно. Решение задач данной группы требует выполнения операции *перебора*, к примеру, некоторой опорной таблицы, или ведения анализа ситуации на поле Робота.
- Примерами задач этого класса могут быть такие:
  - Робот стоит перед входом в длинный «правый» коридор, завершаемый тупиком. В коридоре имеется множество закрашенных и радиационно загрязненных клеток. Записать в таблицу уровни радиации первых ***n*** закрашенных клеток;
  - заполнить таблицу ***b*** значениями тех элементов таблицы ***a***, индексы которых кратны ***3***, но не кратны ***7*** и т.д.

6.2 Задачи на анализ табличных величин

- В задачах этой группы, имея информацию, записанную в табличной величине, найти некоторые ее (табличной величины) характеристики.
- Типичными примерами задач этого класса могут быть такие:
  - нахождение суммы элементов таблицы;
  - подсчет количества элементов таблицы, обладающих заданным свойством (например, положительных/отрицательных/нечетных положительных и т.д.);
  - ответа на вопрос: «сколько пар упорядоченных по возрастанию/убыванию элементов имеется в таблице»?
  - определение минимального/максимального элемента.
- Отметим, что последнюю из названных задач можно в равной мере отнести к следующей группе методов *поиска*. Но согласно нашей классификации, она относится к данной группе *анализа*, поскольку характеризует исследуемую ***таблицу в целом.*** В данном случае поиск ведется *по относительному признаку* (меньше/больше других). Поэтому найти искомую характеристику таблицы можно лишь *при условии полного просмотра всех ее элементов*.
- В силу сказанного, заголовок подобных алгоритмов в общем случае может выглядеть следующим образом:

```
| Образец построения заголовка подпрограммы
| на анализ линейного массива
| в обозначениях МегаЕ

алг тип сумма элементов(арг цел n, арг тип таб a[1 : n])
```

Рис. 6. 7 Заголовок функции типовой задачи на анализ элементов вектора (МегаЕ)

```
| Образец построения заголовка подпрограммы
| на анализ линейного массива
| в обозначениях Delphi
|   где
|     type
|       vector = array of integer;

function summaElementov(const a : vector) : integer;
```

Рис. 6. 8 Заголовок функции типовой задачи на анализ элементов вектора (Delphi)

```
| Образец построения заголовка подпрограммы
| на анализ линейного массива
| в обозначениях C#

int суммаЭлементов(int[] a)
```

Рис. 6. 9 Заголовок функции типовой задачи на анализ элементов вектора (C#)

- По аналогии с предыдущим пунктом, заметим, что в конкретных ситуациях в заголовке могут появляться дополнительные формальные параметры, а вид вспомогательного алгоритма может быть и «процедурный».

6.3 Задачи на поиск в табличных величинах

- Основной вопрос задач этой группы: *имея информацию, записанную в табличной величине, найти ее элемент, обладающий нужным (указанным) свойством*.
- При этом указываемое в условии задачи *свойство должно быть абсолютным*, т.е. для определения пригодности искомого элемента достаточно знать только этот элемент.
- Типичными примерами задач этого класса могут быть такие:
  - найти в таблице элемент с заданным значением;
  - найти в таблице последний/предпоследний/k-й элемент с заданным значением и т.д.

- Отметим следующее обстоятельство: если о таблице (значениях ее элементов) нет никакой дополнительной информации, то поиск нужного элемента можно вести только *последовательным перебором*.
- Если же известно, что значения компонентов таблицы упорядочены, то поиск нужного элемента можно вести более эффективно (быстрее), используя *бинарный* или *двоичный поиск*.

```
| Образец построения заголовка подпрограммы
| на поиск в линейном массиве
| в обозначениях МегаЕ

алг тип максимальный элемент(арг цел n, арг тип таб a[1 : n])
```

Рис. 6. 10 Заголовок функции задачи на поиск максимального элемента вектора (МегаЕ)

```
| Образец построения заголовка подпрограммы
| на поиск в линейном массиве
| в обозначениях Delphi
|   где
|     type
|       vector = array of integer;

function maxElement(const a : vector) : integer;
```

Рис. 6. 11 Заголовок функции задачи на поиск максимального элемента вектора (Delphi)

```
| Образец построения заголовка подпрограммы
| на поиск в линейном массиве
| в обозначениях C#

int максимальныйЭлемент(int[] a)
```

Рис. 6. 12 Заголовок функции задачи на поиск максимального элемента вектора (C#)

6.4 Задачи на перестановки в табличных величинах

- В задачах этой группы, *имея информацию, записанную в табличной величине, требуется поменять местами (*для соблюдения некоторого свойства взаимного расположения*) отдельные ее элементы.*
- Наиболее интересными задачами этой группы являются *задачи сортировки (*упорядочения*) элементов таблицы, без использования дополнительной памяти* (дополнительных таблиц).
- В силу сказанного, заголовок подобных алгоритмов в общем случае может выглядеть следующим образом:

```
| Образец построения заголовка подпрограммы
| на преобразование линейного массива
| в обозначениях МегаЕ

алг сортировка таблицы(арг цел n, арг рез тип таб a[1 : n])
```

Рис. 6. 13 Заголовок функции задачи на сортировку элементов вектора (МегаЕ)

```
| Образец построения заголовка подпрограммы
| на преобразование линейного массива
| в обозначениях Delphi
|   где
|     type
|       vector = array of integer;

procedure sortArray(var a : vector);
```

Рис. 6. 14 Заголовок функции задачи на сортировку элементов вектора (Delphi)

```
| Образец построения заголовка подпрограммы
| на преобразование линейного массива
| в обозначениях C#

void сортировкаМассива(ref int[] a)
```

Рис. 6. 15 Заголовок функции задачи на сортировку элементов вектора (C#)

- В связи с этим, для иллюстрации приемов работы с табличной величиной рассмотрим «комбинированную» задачу следующего содержания:

  Задача 7 Сформировать линейную табличную величину из произвольных целочисленных значений промежутка чисел [-500, 500) и отсортировать ее в порядке неубывания методом просеивания (*челночная сортировка*).

Спецификация ввода: *n*
Спецификация вывода: *последовательность значений элементов*

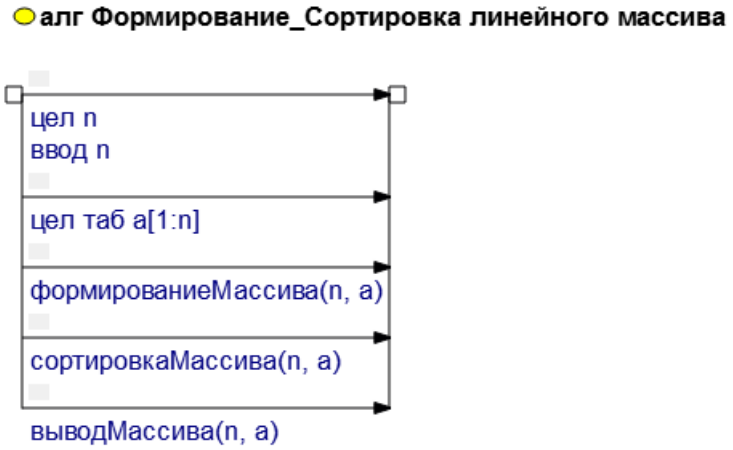
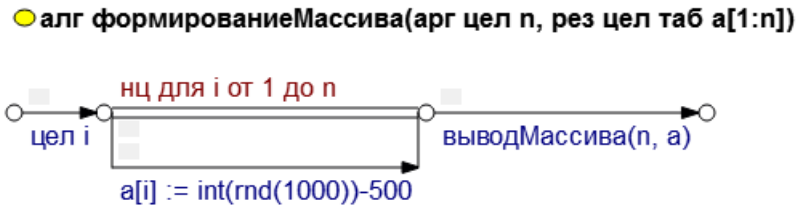


Рис. 6. 16 Описание головной подпрограммы решения задачи (МегаЕ)



Возможное содержимое исходного массива:
206 359 299 -307 -480 -451 337 -49 -450 16

Рис. 6. 17 Детализация вспомогательного алгоритма заполнения массива числами (МегаЕ)

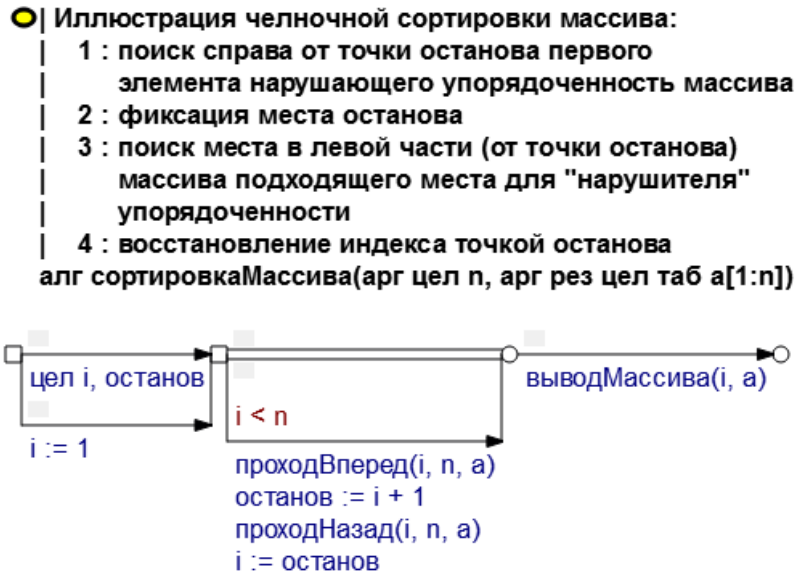


Рис. 6. 18 Детализация головного вспомогательного алгоритма сортировки (МегаЕ)

```
| Поиск "нарушителя" упорядоченности по неубыванию

алг проходВперед(арг рез цел i, арг цел n, арг рез цел a[1:n])
```

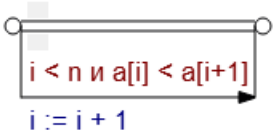


Рис. 6. 19 Детализация вспомогательного алгоритма прямого прохода (МегаЕ)

Первый выход из данного вспомогательного алгоритма произойдет при i = 3, т.к. элемент a[3] = 299 «нарушает» упорядоченность:

206 359 299 -307 -480 -451 337 -49 -450 16

Второй выход произойдет при i = 4, т.к. элемент a[4] = -307 «нарушает» упорядоченность:

206 299 359 -307 -480 -451 337 -49 -450 16

```
| Поиск первого справа-налево места в левой части массива,
| в которое можно поместить очередного "нарушителя"
| упорядоченности.
|   После отыскания этого места производится последовательный
| обмен местами соседних элементов левой части со сдвигом "вправо"
| Другими словами: элементы левой части от найденного места
| сдвигаются вправо на один элемент, а "нарушитель" записывается
| в образовавшуюся "щель"-элемент

алг проходНазад(арг рез цел i, арг цел n, арг рез цел таб a[1:n])
```

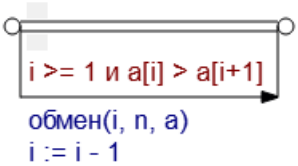


Рис. 6. 20 Детализация вспомогательного алгоритма установки элемента (МегаЕ)

Первый выход из данного вспомогательного алгоритма произойдет после обмена значениями элементов a[2] ↔ a[3]:

206 299 359 -307 -480 -451 337 -49 -450 16,

т.к. значение a[2] после перестановки оказалось больше a[1].

Заметим, что в ходе операций обмена после второго выхода из того же алгоритма перестановки последовательность будет такого вида:

206 299 -307 ↔ 359 -480 -451 337 -49 -450 16

206 -307 ↔ 299 359 -480 -451 337 -49 -450 16

-307 ↔ 206 299 359 -480 -451 337 -49 -450 16

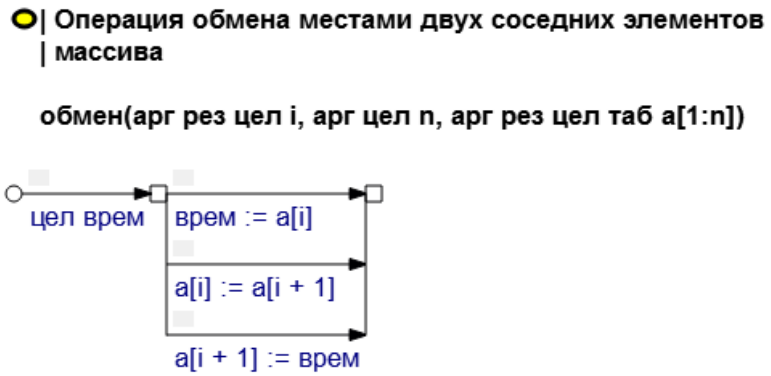


Рис. 6. 21 Детализация вспомогательного обменного алгоритма (МегаЕ)

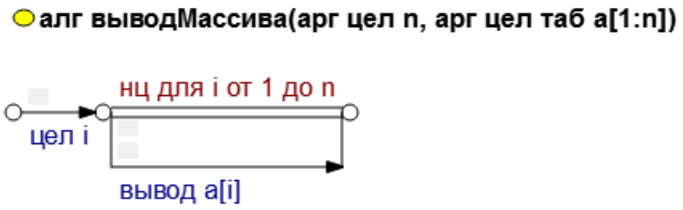


Рис. 6. 22 Детализация вспомогательного алгоритма вывода (МегаЕ)

- Задачи на обработку прямоугольных таблиц фактически решаются совершенно аналогичным способом, что и при обработке линейных таблиц.
- Единственная их особенность – необходимость «управления» двумя индексами. Для этой цели строятся так называемые вложенные циклы. Продемонстрируем это на примере решения следующей задачи.

Задача 8 Дана квадратная таблица **вещ** **таб** A[1 : n, 1 : n]. Необходимо «*исключить*» (занулить) все элементы первого столбца этой таблицы, кроме первого элемента первой строки, используя следующие преобразования:

$$A_{ij} = A_{ij} - A_{i1} \cdot \frac{A_{1j}}{A_{11}}$$

Спецификация ввода: *n*

последовательность элементов таблицы по строкам.

Спецификация вывода:

последовательность элементов таблицы по строкам.

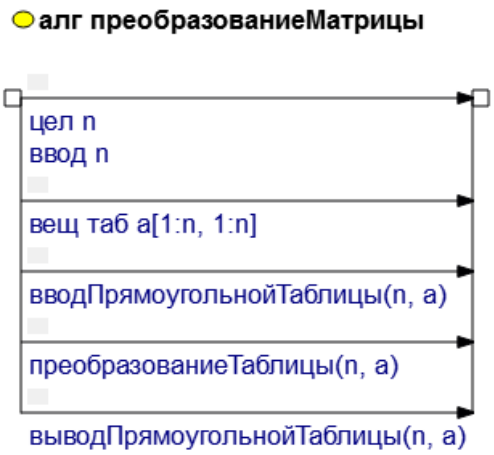


Рис. 6. 23 Головной алгоритм решения задачи 8 (МегаЕ)

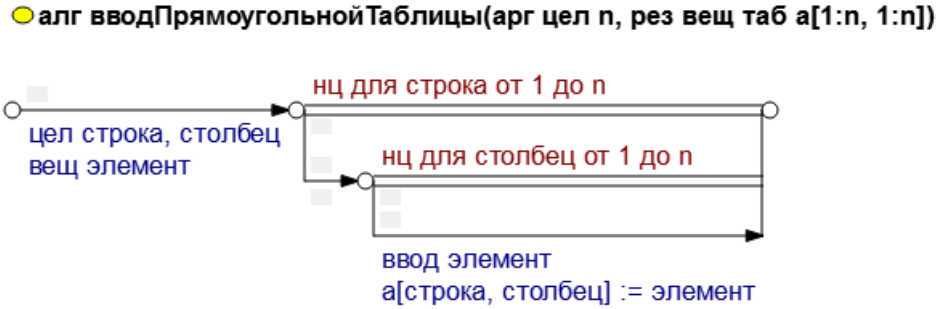


Рис. 6. 24 Детализация вспомогательного алгоритма ввода матрицы (МегаЕ)

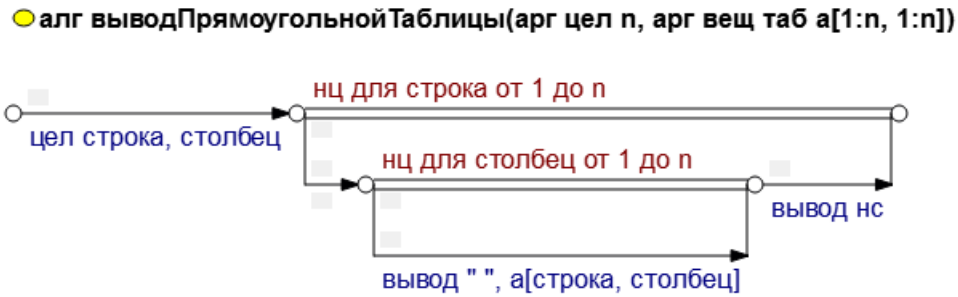


Рис. 6. 25 Детализация вспомогательного алгоритма вывода матрицы (МегаЕ)

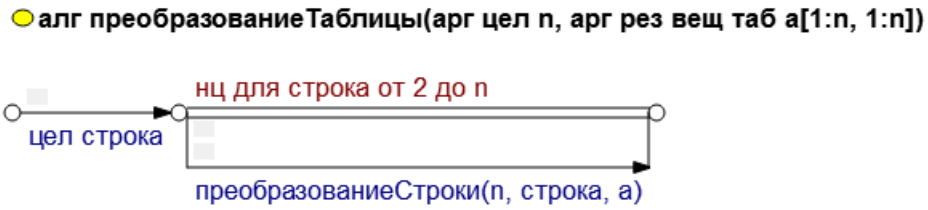


Рис. 6. 26 Детализация вспомогательного алгоритма преобразования матрицы (МегаЕ)

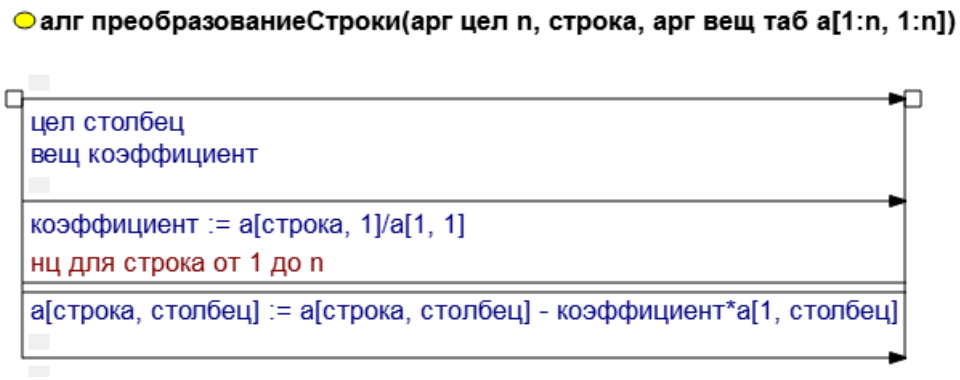


Рис. 6. 27 Детали вспомогательного алгоритма преобразования строки матрицы (МегаЕ)

Задача 9 Найти индекс элемента линейной табличной величины вещественных значений, наименее отличного от среднего арифметического значения всех ее элементов.

Спецификация ввода: n,
значения элементов табличной величины.

Спецификация вывода:
значение индекса.

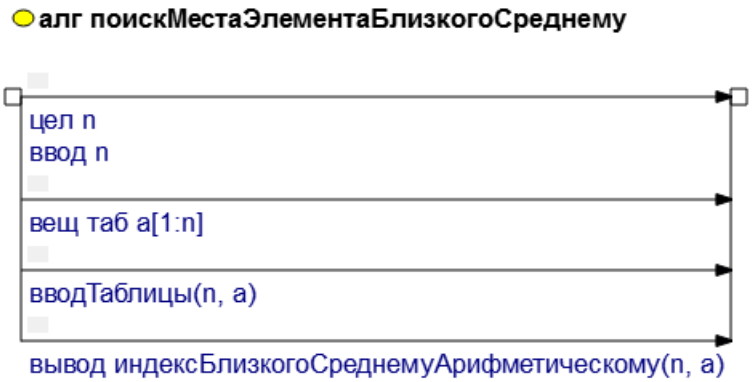


Рис. 6. 28 Головной алгоритм решения задачи 9 (МегаЕ)

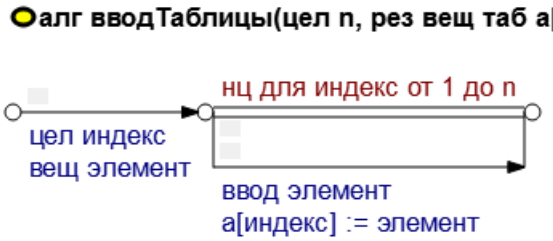


Рис. 6. 29 Детали вспомогательного алгоритма ввода линейной таблицы (МегаЕ)

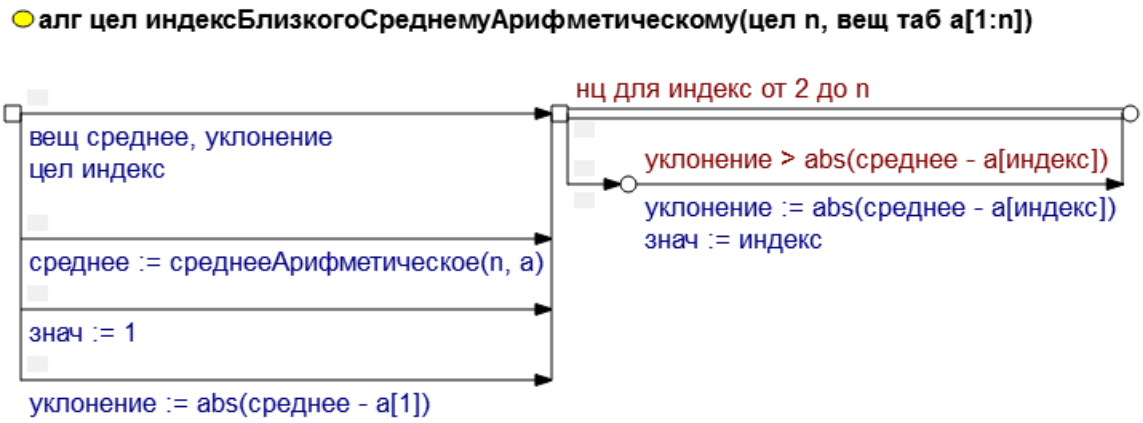


Рис. 6. 30 Детали вспомогательного алгоритма поиска индекса (МегаЕ)

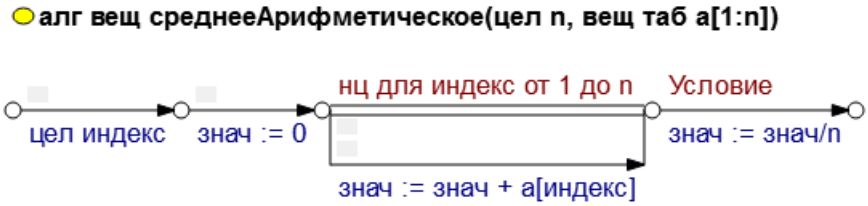


Рис. 6. 31 Детали вспомогательного алгоритма поиска среднего арифметического (МегаЕ)

Задача 10 Составить алгоритм определения, упорядочены ли элементы линейной табличной величины целочисленных значений**.**

Спецификация ввода: n,
значения элементов табличной величины

Спецификация вывода: да или ***нет***

Рекомендация: *Задачу 10 предлагается решить самостоятельно.*

К оглавлению К следующему параграфу К предыдущему параграфу

§7. КЛАССИФИКАЦИЯ ЦИКЛИЧЕСКИХ ПРОЦЕССОВ ОБРАБОТКИ СТРОКОВЫХ ВЕЛИЧИН[21]

- Учитывая большое сходство в организации и обработке табличных и литерных величин, в данном случае, следуя п. 3.6, выделим те же группы задач:

Рис. 7. 1 Классификация задач по обработке строковых величин

- Сходство между линейными таблицами и литерными величинами в следующем:

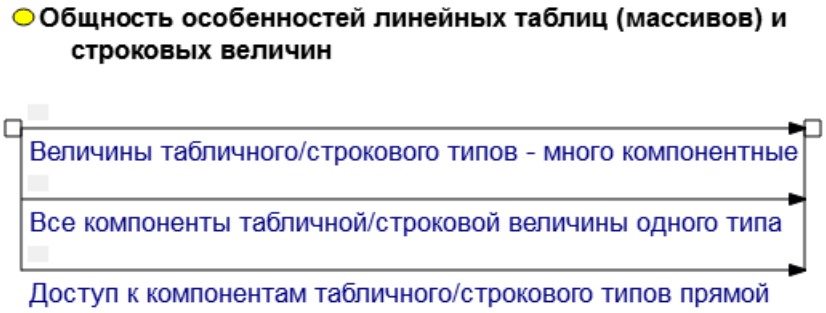


Рис. 7. 2 Общность задач

- Вместе с тем, у литерных величин имеются и свои особенности, которые позволяют их обрабатывать по-особому. Эти особенности таковы:

[21] Ознакомиться с иллюстрирующими Р-схемами данного параграфа можно в документе 03, папки 03, «03 Обработка строковых величин» и 04 – «04 Образцы решения задач на обработку строк» <Док. 03.03-04>

Рис. 7. 3 Характерные особенности задач

- вместе с тем, что литерная величина может содержать множество компонент, синтаксисом допустимо, чтобы это множество было *пустым*. Этот случай инициируется командой присваивания вида:

  Текст :=””, где переменная ***Текст*** описана так: **лит** Текст;

- единственным простым типом элементов литерной величины может быть только символьный (**сим**) ;
- над множеством литерных значений определены следующие операции:
  - – операция *вырезки*: **Текст[i : j]** – часть литерной величины (*подстрока*) от ***i***-символа до ***j***-символа, включая их самих. Понятно, что значения параметров ***i*** и ***j*** должны удовлетворять следующему неравенству:

    1 ≤ i ≤ j ≤ количество элементов литерной величины;

  - – операция *склеивания*: **Текст1 + Текст2** – новый текст, получаемый приписываем к элементам первой величины (без отступа) элементов второй. И в данном случае существует одно ограничение: суммарное количество элементов нового (склеиваемого) текста будет не больше 255 символов. Если реальная ситуация такова, что это условие нарушается, то «лишние» символы, начиная с 256 – отбрасываются;
  - – имеется функция: ***длин*(Текст)** – результатом действия которой является выдаваемое ею количество элементов, содержащихся в литерной величине Текст.

▪ Множество задач группы «*на заполнение*» традиционно сводится в данном случае к группе *заполнение с поиском*.

- Дело в том, что «строить текст» с использованием, например, датчика случайных чисел практически лишено смысла в силу невозможности получения «связного» текста.
- Все остальные группы задач имеют те же особенности, что и в случае обработки линейных табличных величин. Это показывает и решение приводимых ниже задач.

Задача 11 Сформировать литерную величину из «серединных» букв слов данной литерной величины *Текст*.

Определение: Слово – это непустая последовательность символов текста.

Замечание: Если обрабатываемое слово содержит четное число букв, в качестве символа-вставки в литерную величину ответа использовать пробел.

Спецификация ввода: ***Текст***
Спецификация вывода: Сформированный текст

Задача 11
Сформировать литерную величину из «серединных» букв слов, данной литерной величины Текст.

Определение: Слово – это непустая последовательность символов текста.

Замечание: Если обрабатываемое слово содержит четное число букв, в качестве символа-вставки в литерную величину ответа использовать пробел.

Спецификация ввода :
Текст
Спецификация вывода :
сформированный текст

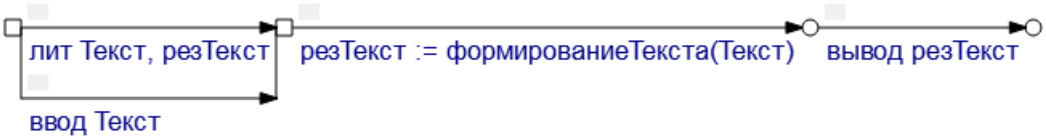


Рис. 7. 4 Головной алгоритм решения задачи на заполнение

алг лит формированиеТекста(лит Текст)

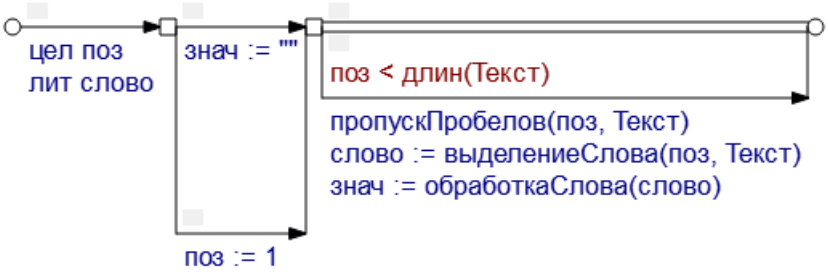


Рис. 7. 5 Основной вспомогательный алгоритм формирования текста

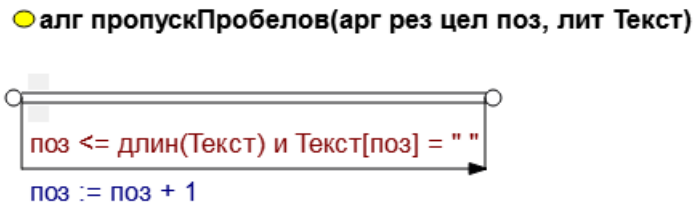


Рис. 7. 6 Вспомогательный алгоритм пропуска пробелов в тексте

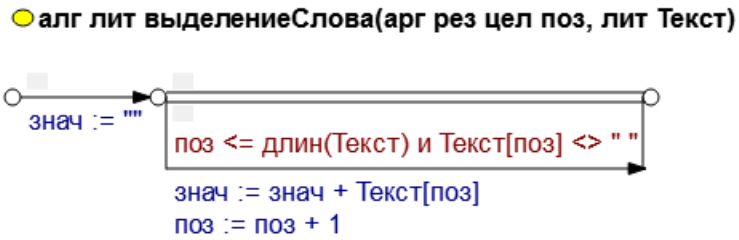


Рис. 7. 7 Вспомогательный алгоритм выделения слова из текста

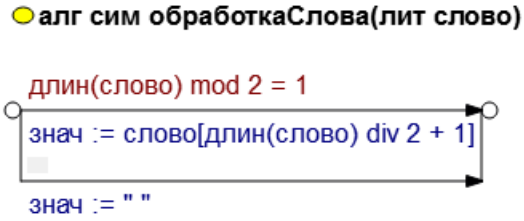


Рис. 7. 8 Вспомогательный алгоритм определения серединного символа слова

Задача 12 Дана литерная величина ***Текст.*** Определить количество слов, входящих в ее состав.

Определение: Слово – это непустая последовательность символов текста.

Спецификация ввода: ***Текст***

Спецификация вывода: количество слов

```
|     Задача 12
| Дана литерная величина Текст.
| Определить количество слов входящих в ее состав.
|
|        Определение: Слово – это непустая последовательность символов текста.
|
|        Спецификация ввода    :
|                Текст
|        Спецификация вывода :
|                количество слов

алг цел количествоСлов(лит Текст)
```

Рис. 7. 9 Вспомогательный алгоритм определения серединного символа слова

```
алг цел количествоСлов(лит Текст)
нач
 . цел поз
 . поз :=1
 . знач := 0
 . нц пока(поз <= длин(Текст))
 . . пропуститьПробелы(поз, Текст)
 . . если (поз <= длин(Текст))
 . . . то
 . . . . знач := знач + 1
 . . . . пропуститьСлово(поз, Текст)
 . . все
 . кц
кон
```

```
алг цел количествоСлов(лит Текст)
{
    цел поз
    поз :=1
    знач := 0
    while(поз <= длин(Текст)){
        пропуститьПробелы(поз, Текст)
        if(поз <= длин(Текст)){
            знач := знач + 1
            пропуститьСлово(поз, Текст)
        }
    }
}
```

Рис. 7. 10 Сравнение алгоритмов по принципу («надо» слева и «получили» (справа))

Задача 13 Дана литерная величина ***Текст*** и символьная величина ***Буква***. Определить длину последнего слова в величине ***Текст***, заканчивающегося на символ ***Буква***.

Определение: Слово – это непустая последовательность символов текста.

Замечание: Если указанного в задаче сделать нельзя, в качестве результата вывести значение «***-1***».

Спецификация ввода: Текст
Буква

Спецификация вывода: *значение длины слова* или ***-1***

Задача 13

Дана литерная величина Текст и символьная величина Буква. Определить длину последнего слова в величине Текст, заканчивающегося на символ Буква.

Определение: Слово – это непустая последовательность символов текста.

Замечание: Если указанного в задаче сделать нельзя, в качестве результата вывести значение «-1».

Спецификация ввода :
Текст (одна строка)
Буква

Спецификация вывода :
значение длины слова или -1

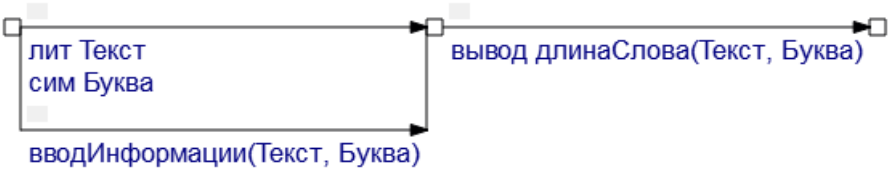


Рис. 7. 11 Оформление головного алгоритма с включением условия

алг вводИнформации(рез лит Текст, рез сим Буква)

Рис. 7. 12 Вспомогательный алгоритм ввода данных

алг цел длинаСлова(лит Текст, сим Буква)

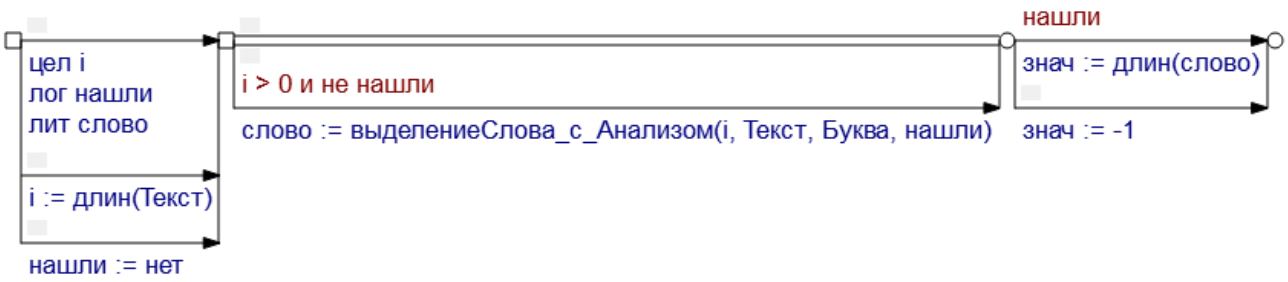


Рис. 7. 13 Основной алгоритм решения задачи

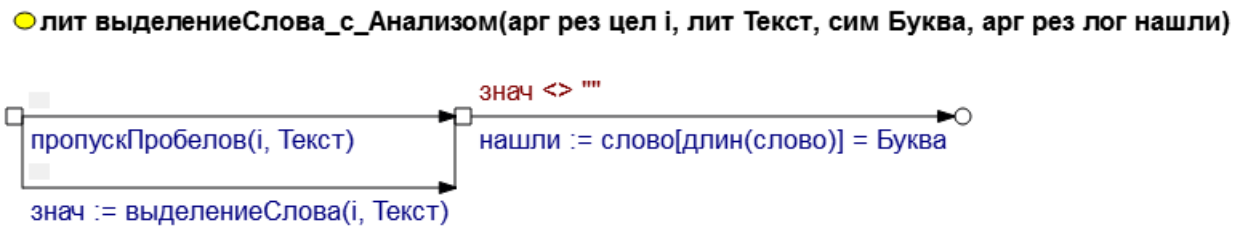


Рис. 7. 14 Вспомогательный алгоритм выделения слова с дополнительным анализом

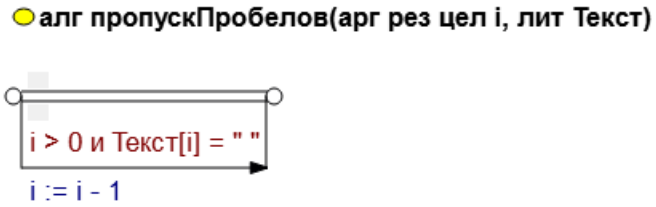


Рис. 7. 15 Вспомогательный алгоритм пропуска пробелов в тексте

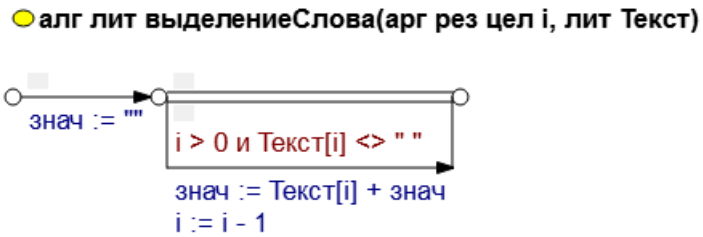


Рис. 7. 16 Вспомогательный алгоритм выделения очередного слова из текста

Задача 14 Дан ***Текст***, слова которого разделены произвольным количеством пробелов. ***Текст*** может начинаться и завершаться тоже произвольным числом пробелов. Надо начальные и конечные пробелы в тексте удалить, а между словами оставить по одному пробелу.

Определение: Слово – это непустая последовательность символов текста.

Спецификация ввода: ***Текст***

Спецификация вывода: преобразованное значение величины ***Текст***

Задача 14

Дан Текст, слова которого разделены произвольным количеством пробелов. Текст может начинаться и завершаться тоже произвольным числом пробелов. Надо начальные и конечные пробелы в тексте удалить, а между словами оставить по одному пробелу.

Определение: Слово – это непустая последовательность символов текста.

Спецификация ввода :
Текст
Спецификация вывода :
преобразованное значение величины Текст

Рис. 7. 17 Оформление головного алгоритма с включением условия

алг удаленияНачальныхПробелов(арг рез лит Текст)

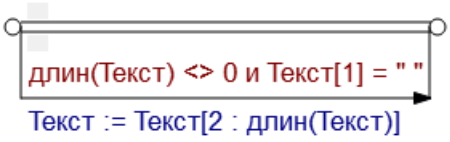


Рис. 7. 18 Оформление вспомогательного алгоритма удаления

алг обработкаТекста(арг рез лит Текст)

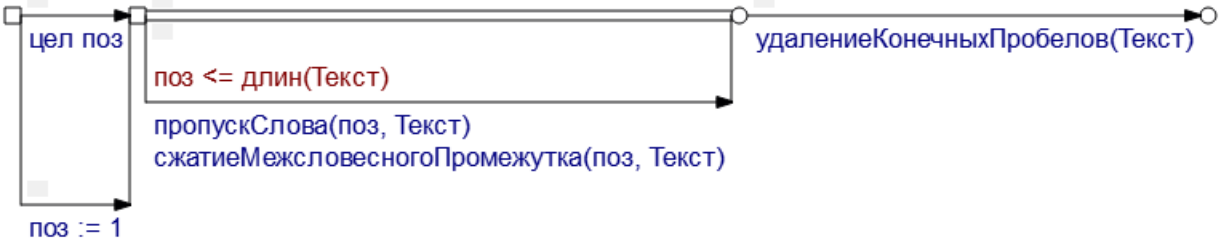


Рис. 7. 19 Оформление вспомогательного алгоритма обработки середины текста

алг сжатиеМежсловесногоПромежутка(арг рез цел поз, арг рез лит Текст)

Рис. 7. 20 Оформление вспомогательного алгоритма сжатия пробела между словами

```
алг пропускСлова(арг рез цел поз, лит Текст)

нц пока поз <= длин(Текст) и Текст[поз] <> " "
    поз := поз + 1
```

Рис. 7. 21 Вспомогательный алгоритм пропуска слова

```
алг пропускПробелов(арг рез цел поз, лит Текст)

нц пока поз <= длин(Текст) и Текст[поз] = " "
    поз := поз + 1
```

Рис. 7. 22 Вспомогательный алгоритм пропуска пробелов

```
алг удалениеКонечныхПробелов(арг рез лит Текст)

нц пока длин(Текст) <> 0 и Текст[длин(Текст)] = " "
    Текст := Текст[1 : длин(Текст)-1]
```

Рис. 7. 23 Вспомогательный алгоритм удаления пробелов в конце текста

К оглавлению К следующему параграфу К предыдущему параграфу

§8. Базовые схемы построения циклов[22]

8.1 Предварительные замечания

- Множество управляющих структур любого алгоритмического языка непременно включает в свой состав некоторое количество команд по организации циклов:

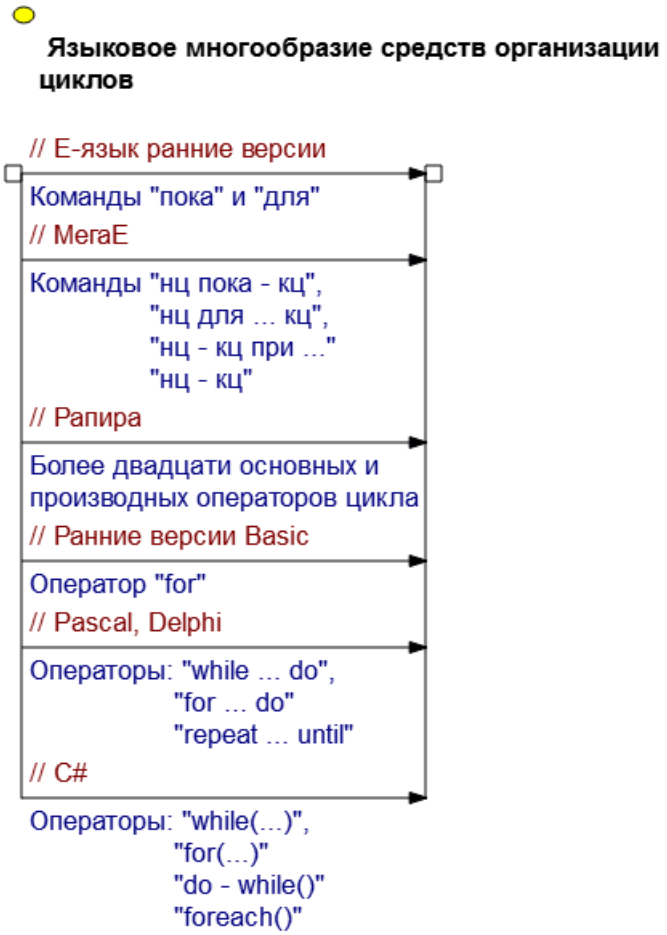


Рис. 8. 1 Операторы (команды) цикла и языки программирования

- Ситуации, в которых приходится прибегать к составлению циклических алгоритмов, достаточно часты (99% от общего количества решаемых задач), поэтому они, естественно, не исчерпываются только использованием простых операторов цикла («голые» операторы цикла и 1-2 простых оператора в его рабочей части).
- В практике программирования рассматриваются определенные приемы конструирования циклов за счет их сведения к так называемым, *базовым схемам циклической обработки информации*.

[22] Ознакомиться с иллюстрирующими Р-схемами данного параграфа можно в документе 04, папки 01-02, «01 Введение» и «02 Рекурсивная схема» <Док. 04.01-02>

- Таких схем различают несколько:

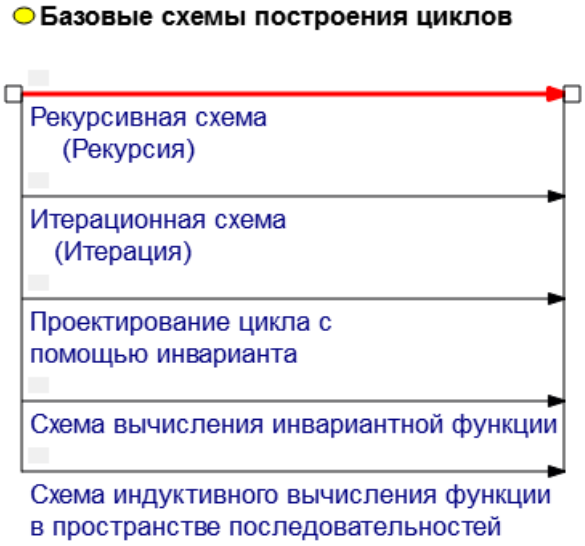


Рис. 8. 2 Базовые схемы построения циклов

8.2 Рекурсивная схема построения циклов

8.2.1 Определения и пример

- Понятия *рекурсии*, *рекурсивных алгоритмов* тесно связаны с понятиями *рекуррентных* соотношений, формул, определений и так далее.
- Заметим, что *recurrence* в переводе с английского означает – возвращение, повторение.
- Обычно под рекуррентной или рекурсивной формулой или определением понимают такие формулы или определения, в определяющей части которых встречается определяемое понятие.
- Приведем некоторые примеры.
  - Определение факториала числа:

$$n! = \begin{cases} 1, & n = 0, \\ n*(n-1)!, & n > 0. \end{cases}$$

  - Понятие чисел Фибоначчи:

$$f_n = \begin{cases} 1, & n = 0, \\ 1, & n = 1, \\ f_{n-1} + f_{n-2}, & n \geq 2. \end{cases}$$

  - Определение понятия *идентификатор* средствами РБНФ строилось так:

    Имя = *Буква*{*Буква*|*Цифра*}

    Заметим, однако, что приведённое определение равносильно такому:

Имя = [Буква| Имя Буква| Имя Цифра]

Рис. 8. 3 Определение понятия «Идентификатор» («Имя»)

Определение Рекурсия – это такая схема обработки информации, при которой программа вызывает саму себя

- либо непосредственно (*прямая рекурсия*);
- либо через посредство других программ (*косвенная рекурсия*).

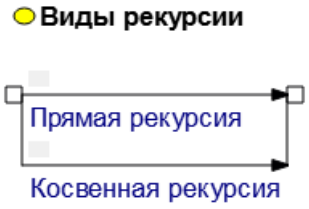


Рис. 8. 4 Виды рекурсивных обращений

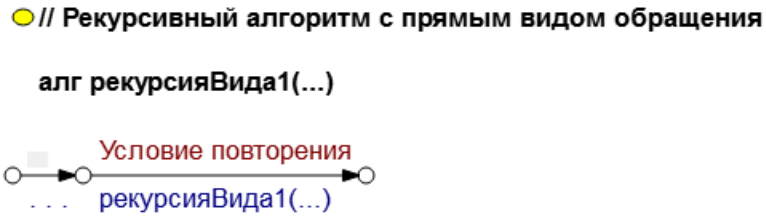


Рис. 8. 5 Р-схема прямого рекурсивного обращения

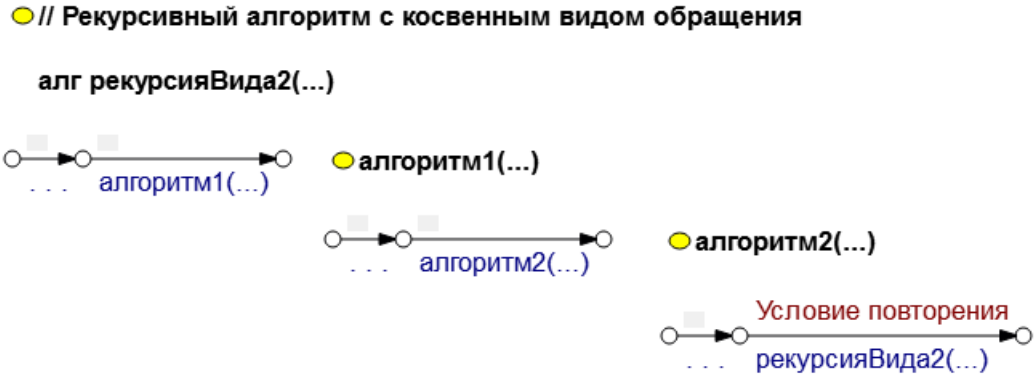


Рис. 8. 6 Р-схемы косвенного рекурсивного обращения

- Суть рекурсии – в сведении подзадачи, возможно, лишь части общей решаемой задачи, к другой подзадаче такого же вида, но с другими исходными данными.

8.2.2 Требования, предъявляемые к рекурсивным алгоритмам

- Сформулируем теперь основные требования, которые необходимо предъявлять к рекурсивному алгоритму, чтобы уберечь его от грубых логических ошибок.

  Во-первых, рекурсивный алгоритм должен удовлетворять свойству *конечности*. Реализация этого свойства обычно осуществляется за счет *выделения* в *алгоритме не менее одного такого частного случая* проведения расчетов, в котором искомое *значение считается непосредственно*, то есть, не прибегая к рекурсивному обращению.

  Во-вторых, в алгоритме должна присутствовать хотя бы одна *величина* (или *одно состояние)*, которая, *монотонно изменяясь*, обеспечивала бы в конце концов приход к необходимости реализации отмеченного выше (свойство 1) частного (тривиального) случая ведения просчетов или принятия решений. Эта величина/состояние носит название *управляющей величины* или *параметра рекурсии*.

  В-третьих, считается, что задача удобна для построения рекурсивного алгоритма, если она *может быть представлена в виде совокупности подзадач того же типа*, но с иными значениями или состояниями (монотонно иными) параметра рекурсии.

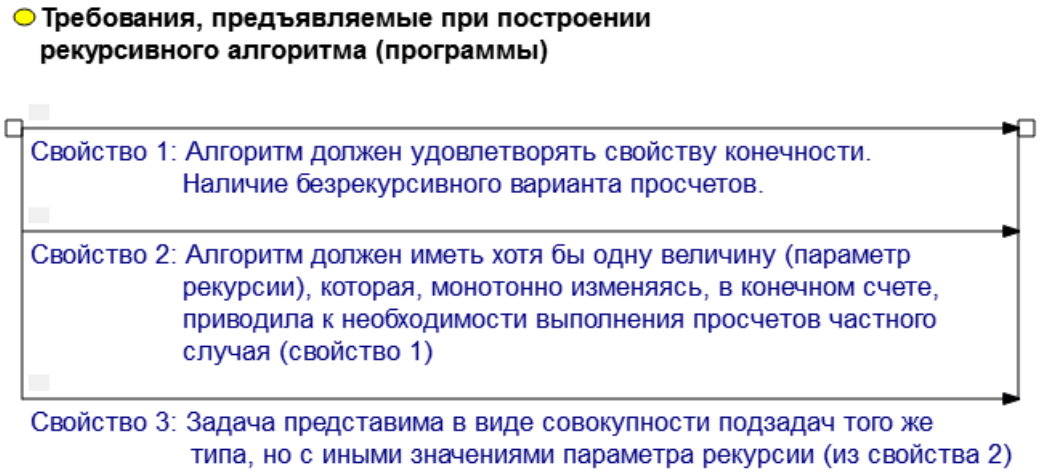


Рис. 8. 7 Требования, которым должен удовлетворять рекурсивный алгоритм

8.2.3 Общая методика построения рекурсивного алгоритма

- При всей неоднозначности подхода к построению рекурсивных программ, можно все-таки выделить следующие три основные момента, непосредственно вытекающие из требований, предъявляемых к рекурсивным алгоритмам. Эти моменты или этапы таковы:

  Во-первых, необходимо провести ***параметризацию задачи***, то есть выделить или определить те величины, те состояния (размерность

задачи, невязка, погрешность, положение исполнителя в его среде обитания и так далее), которые *монотонно изменяются при переходе из одной ситуации решения задачи к другой*.

Во-вторых, надо найти тот ***простейший*** или ***тривиальный случай*** в общем решении задачи, при реализации которого не будет нужды производить рекурсивное обращение (невязка мала по сравнению с контрольной величиной, размерность задачи достигла значения, равного 1 или 0, исполнитель находится в некотором предельно допустимом положении: Робот – у стены, Путник – у препятствия и так далее).

В-третьих, надо ***правильно*** организовать ***декомпозицию*** общего случая решения задачи. Цель этого момента – привести алгоритм к необходимости решения одной или нескольких новых задач, аналогичных исходной, но с другими значениями параметров (новым значением невязки или погрешности, новой размерностью, новым положением исполнителя в среде и так далее) по сравнению с данным и предыдущими шагами.

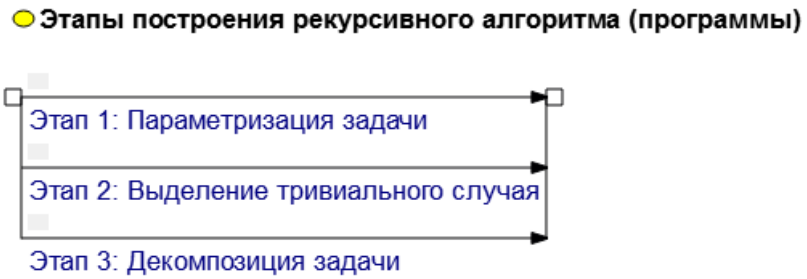


Рис. 8. 8 Этапы построения рекурсивного алгоритма

- Проведем иллюстрацию предложенной методики решения задач на классическом примере просчета факториала натурального числа ***n***.

Задача 15. Составить рекурсивный алгоритм просчета значения ***n!***.

Определение:

$$n! = \begin{cases} 1, & n = 0, \\ n*(n-1)!, & n > 0. \end{cases}$$

Спецификация ввода: ***n***
Спецификация вывода: значение n!

- ***<u>Этап 1</u>***. Естественным и единственным параметром в этом примере является переменная ***n***. После чего мы можем оформить заголовок рекурсивного алгоритма-функции (почему именно функции ?) так:

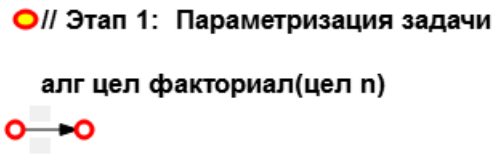


Рис. 8. 9 Результат первого этапа построения алгоритма

- ***<u>Этап 2</u>***. Определение тривиального случая в данном примере тоже не вызывает особых трудностей, так как он просматривается из самого определения понятия *факториал*, приведенного выше. Тривиальным случаем является ситуация, когда значение параметра ***n*** принимает значение ***0***, так как в этом случае факториал полагается *равным* ***1***. В итоге можем записать

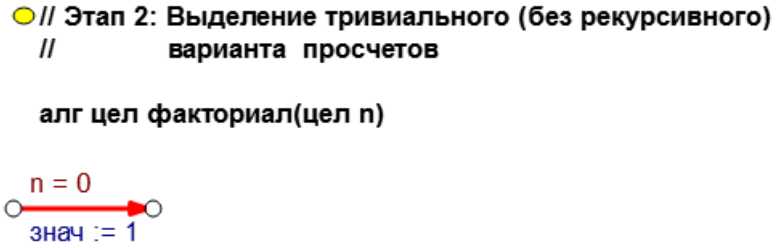


Рис. 8. 10 Результат второго этапа построения алгоритма

- ***<u>Этап 3</u>***. Опять же, из определения факториала видно, как произвести декомпозицию или редукцию исходной задачи к аналогичной задаче, но меньшей размерности (по крайней мере, на единицу). Этот шаг описывается в серии команд 2 команды «*если*» следующим образом:

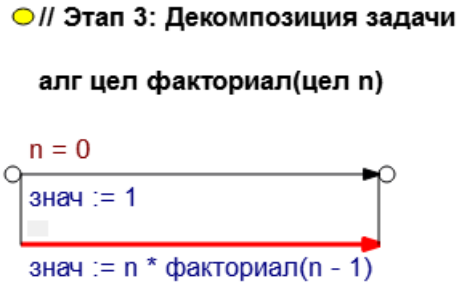


Рис. 8. 11 Результат третьего этапа построения алгоритма

В итоге получаем рекурсивный алгоритм вида:

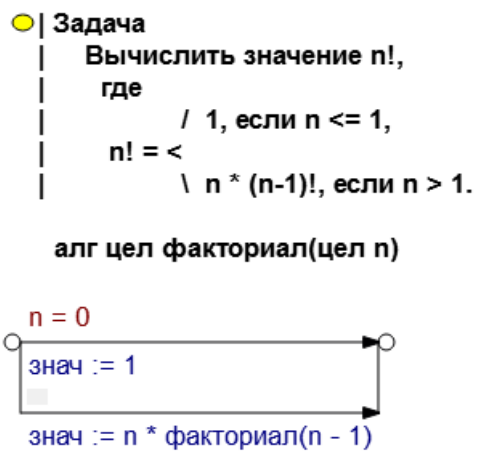


Рис. 8. 12 Итоговая Р-схема решения задачи

- Отметим основные достоинства и недостатки использования рекурсивных алгоритмов в практике программирования.

8.2.4 Достоинства и недостатки схемы

Достоинства:

- Данное средство позволяет избрать иную, по сравнению с традиционной, систему базовых управляющих структур структурной технологии программирования. То есть вместо совокупности базовых структур управления *ПРИСВАИВАНИЕ*, *ВЕТВЛЕНИЕ* и *ЦИКЛ* можно использовать равную ей по выразительности совокупность *ПРИСВАИВАНИЕ*, *ВЕТВЛЕНИЕ* и *РЕКУРСИЯ*.
- Данное средство во многих случаях позволяет сохранить при решении задачи её естественную постановку, как это было в случае просчета значения факториала. Дело в том, что практически ни в одной постановке задач явно не фигурирует указание на использование циклов. Этому приходится обучаться. Рекурсия же позволяет в некоторых случаях сохранить естественность решения.

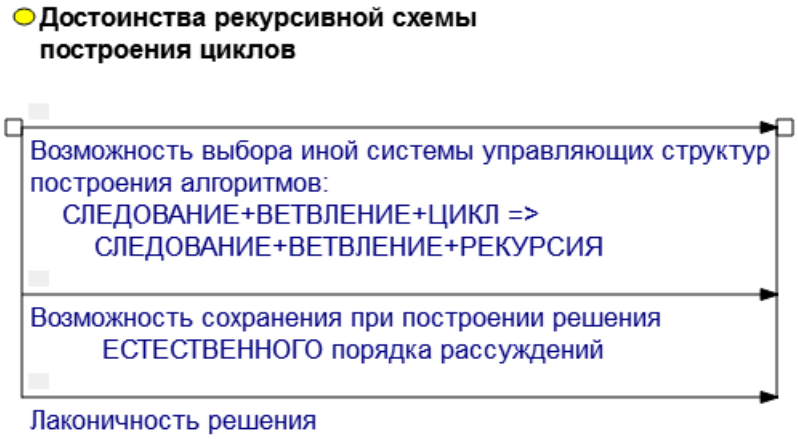


Рис. 8. 13 Перечень основных достоинств рекурсивной схемы

Недостатки:

- Ограниченность стековой памяти компьютера может послужить естественным барьером в решении отдельных задач, где количество циклических обработок фрагментов программ будет превосходить (даже хотя бы на 1) предельно допустимую глубину стека.
- Неаккуратное построение рекурсивных алгоритмов нередко приводит к зацикливанию с выдачей сообщения о переполнении стека, то есть с аварийным остановом по ошибке. Кстати, подобная же ошибка с зацикливанием при использовании операторов цикла «переживается» пользователями почему-то менее болезненно.
- Непродуманное использование рекурсивных обращений может служить источником неоправданно больших временных затрат при решении отдельных задач. Пример с нахождением значений чисел Фибоначчи.

Рис. 8. 14 Перечень основных недостатков рекурсивной схемы

8.2.5 Семантика обработки рекурсивных алгоритмов. Формуляры

- С целью уяснения семантики обработки рекурсивных алгоритмов Гооз и Бауэр[23] предлагают использовать аппарат формуляров, построенный на аппарате деревьев Канторовича.
- Для упрощения графических изображений будем использовать средства, представляющие собой стилизованные деревья Канторовича.
- Обращение к вспомогательному алгоритму *факториал* описывается стилизованным формуляром вида:

Факториал = _____
n = _____

Рис. 8. 15 Общий (начальный) формуляр алгоритма «Факториал»

- Развернутая форма стилизованного формуляра для рассмотренного выше примера будет иметь вид:

[23] Бауэр Ф.Л., Гооз Г. Информатика. Вводный курс: в 2-х ч. Ч.2. Пер. с нем. – М.: Мир, 1990, с.560-563

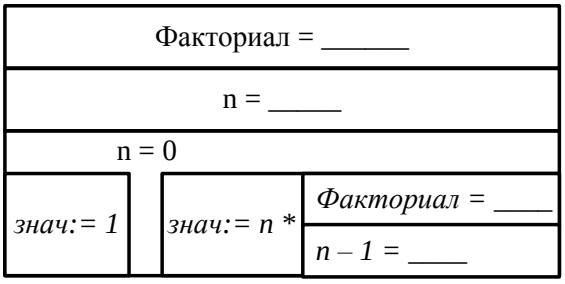


Рис. 8. 16 Детализированный формуляр алгоритма «Факториал»

- Пусть, для определённости, надо просчитать значение ***3!*** Используя предложенное средство, рассмотрим семантику обработки рекурсивного алгоритма *факториал(3)*. Получим следующую последовательность формуляров:

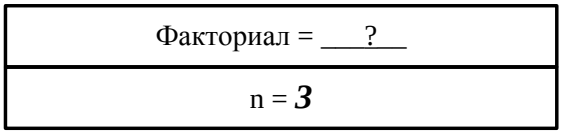


Рис. 8. 17 Общий, исходный формуляр алгоритма «Факториал» с параметром n = 3

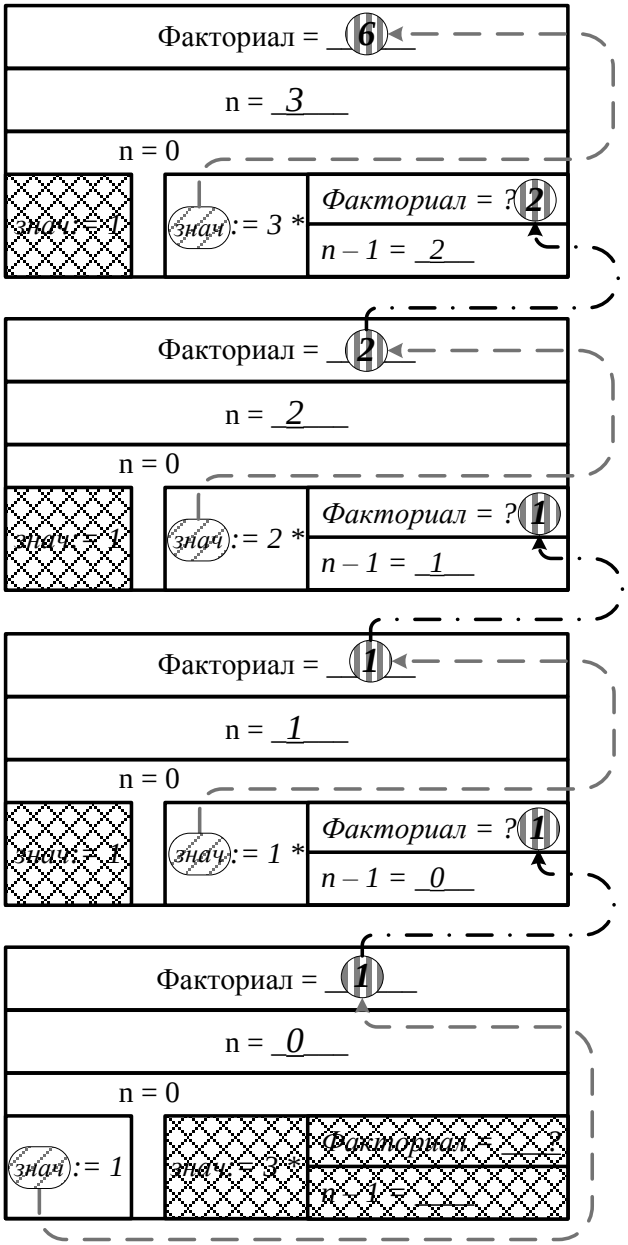


Рис. 8. 18 Последовательность детализаций формуляра

- Средствами Excel-таблиц можно получить «действующую» модель формуляров.

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	*Зеленая область - это область копирования неизменяемой части; Фиолетовая часть - это "размножаемая" таблица для копирования; Процесс копирования продолжается до тех пор, пока в контролируемой части появится значение ИСТИНА*												
2													
3		Общий формуляр метода											
4		факториал =	120										
5		n =	5										
6													
7		факториал =	120								Формуляр №	1	
8		n =	5										
9			n (	5	) = 0	ЛОЖЬ							
10		return		?		return	n (	5	) *	факториал =	24	120	
11										n-1 =	4		
12													
13		факториал =	24								Формуляр №	2	
14		n =	4										
15			n (	4	) = 0	ЛОЖЬ							
16		return		?		return	n (	4	) *	факториал =	6	24	
17										n-1 =	3		
18													

Рис. 8. 19 Заготовки формуляров (с полученными результатами).

	A	B	C	D	E	F	G	H	I	J	K
1						Заготовки					
2											
3	факториал	0									
4	n =	3									
5											
6	факториал	0								Формуляр №	1
7	n =	3									
8		n (	3	) = 0	ЛОЖЬ						
9	return		?		return	n (	3	) *	факториал	0	0
10									n-1 =	2	

Рис. 8. 20 Заготовки формуляров (со статичным заполнением).

B3 :	=B6		
B4 :	< Вносится нужное число>		
B6 :	=ЕСЛИ(E8;C9;K9)	K6 :	1
B7 :	=B4		
C8 :	=B7	E8 :	=C8=0
C9 :	=ЕСЛИ(E8;1;"?")	J9 :	=B12
G9 :	=ЕСЛИ(E8;"?";C8)	J10 :	=ЕСЛИ(E8;"?";C8-1)

K9 :	=ЕСЛИ(E8; "?";G9*J9)

Рис. 8. 21 Формульное содержимое ячеек к рис. 8.20.

	A	B	C	D	E	F	G	H	I	J	K
11											
12	факториал	0								Формуляр №	2
13	n =	2									
14		n (	2	) = 0	ЛОЖЬ						
15	return		?		return	n (	2	) *	факториал	0	0
16									n-1 =	1	
17											

Рис. 8. 22 Заготовка копируемого формуляра

B12 :	=ЕСЛИ(E14;C15;K15)	K12 :	=K6+1		
B13 :	=J10				
C14 :	=B13	E14 :	=C14=0		
C15 :	=ЕСЛИ(E14;1;"?")	J15 :	=B18	K15 :	=ЕСЛИ(E14; "?";G15*J15)
G15	=ЕСЛИ(E14;"?";C14)	J16 :	=ЕСЛИ(E14;"?";C14-1)		

Рис. 8. 23 Формульное содержимое ячеек к рис. 8.22

8.2.6 Примеры написания простых рекурсивных алгоритмов

Реализация аналога А – цикла

Задача 16 Найти значение алгебраического полинома ***p(x),*** где

$$p(x) = x^n + 2\,x^{n-1} + 3\,x^{n-2} + \ldots + n\,x + n+1.$$

Спецификация ввода: ***n x***
Спецификация вывода: *значение полинома*

- *параметром рекурсии* данного типа циклического процесса возьмём ***коэффициент*** *при аргументе* ***x***. Применим при просчёте значения многочлена схему Горнера:

$$x^n + 2\,x^{n-1} + 3\,x^{n-2} + \ldots + n\,x + n+1 = (\ldots((x + 2)*x + 3)*x + \ldots + n)*x + n+1.$$

  Рекурсию будем строить по убывающему значению параметра. Следовательно, рекурсивные обращения будут «разворачиваться от конца к началу».
- *тривиальных случаев* будет два. Первый – это частный случай 0-многочлена. Второй случай – это случай, когда коэффициент становится меньшим 1. Такого члена в многочлене нет, но перед последним рекурсивным обращением была произведена операция умножения на ***x***. Для получения корректного результата в тривиальном случае должен быть инициирован нейтральный элемент по операции «умножение». Таким элементом является 1.
- Этап *декомпозиции задачи* состоит в домножении на ***x*** «*внутренних*» значений скобочных выражений.

```
| Задача 16
|     Построить рекурсивную функцию просчета значения
| алгебраического полинома p(x), где
|     p(x) = x_[n] + 2*x_[n-1] + 3*x_[n-2] + ... + n*x + n+1.

алг вещ схемаГорнера(арг цел коэффициент, вещ x)

коэффициент = 0
знач := 1

знач := значениеПолинома(коэффициент - 1, x)*x + (коэффициент + 1)
```

Рис. 8. 24 Рекурсивная версия реализации А-цикла

- Проанализируем семантическую суть приведенного алгоритма для многочлена конкретного вида: $p(x) = x^4 + 2\,x^3 + 3\,x^2 + 4\,x + 5$, который в «свернутом виде» может быть записан так: $p(x) = (((x + 2)*x + 3)*x + 4)*x + 5$.
- Общий формуляр алгоритма таков:

	A	B
1	*значениеПолинома* =	1
2	коэффициент =	
3	x =	

Рис. 8. 25 Вид общего формуляра

	A	B	C	D	E	F	G	H	I	J	K	L
5	*значениеПолинома* =	1										
6	коэффициент =	0										
7	x =	0										
8	коэффициент (	0	) = 0	ИСТИНА								
9					*значениеПолинома* =	*1*						
10	знач:=	1		знач:=	коэффициент - 1 =	-1	*x(	0	) + (коэффициент + 1) (	1	)	1
11					x =	0						

Рис. 8. 26 Вид детализированного формуляра

- Нисходяще-восходящая последовательность формуляров с информацией о передаче параметров показана на рисунке

	A	B	C	D	E	F	G	H	I	J	K	L
5	*значениеПолинома* =	3.5625										
6	коэффициент =	2										
7	x =	0.25										
8	коэффициент (	2	) = 0	ЛОЖЬ								
9					*значениеПолинома* =	*2.25*						
10	знач:=	?		знач:=	коэффициент - 1 =	1	*x(	0.25	) + (коэффициент + 1) (	3	)	3.5625
11					x =	0.25						
12												
13	*значениеПолинома* =	2.25										
14	коэффициент =	1										
15	x =	0.25										
16	коэффициент (	1	) = 0	ЛОЖЬ								
17					*значениеПолинома* =	*1*						
18	знач:=	?		знач:=	коэффициент - 1 =	0	*x(	0.25	) + (коэффициент + 1) (	2	)	2.25
19					x =	0.25						
20												
21	*значениеПолинома* =	1										
22	коэффициент =	0										
23	x =	0.25										
24	коэффициент (	0	) = 0	ИСТИНА								
25					*значениеПолинома* =	*1*						
26	знач:=	1		знач:=	коэффициент - 1 =	-1	*x(	0.25	) + (коэффициент + 1) (	1	)	1.25
27					x =	0.25						

Рис. 8. 27 Последовательность заполненных детализированных формуляров

B1 :	=ЕСЛИ(B5<>0;B5;"?")
B2 :	< Вносится нужное число>
B3 :	< Вносится нужное число>

Рис. 8. 28 Формульное содержимое ячеек к рис. 8.25.

B5 :	=ЕСЛИ(B10=1;B10; L10)		
B6 :	=B2		
B7 :	=B3		
B8 :	=B6	D8 :	=ЕСЛИ(B8=0;"ИСТИНА";"ЛОЖЬ")
	F9 :	=ЕСЛИ(B10=1;1;B13)	
B10 :	=ЕСЛИ(D8="ИСТИНА";1;"?")		
	F10 :	=B6-1	
	H10 :	=B7	
	J10 :	=B6+1	
	L10 :	=ЕСЛИ(F9="?";"?";F9*H10+J10)	

Рис. 8. 29 Формульное содержимое ячеек к рис. 8.26.

<u>Реализация аналога КМВ – цикла</u>

Задача 17 Найти сумму вида

$$S = 1 + \frac{1}{2!} + \frac{1}{3!} + \ldots = \sum_{k=0} \frac{1}{k!}$$ с точностью ε.

Спецификация ввода: *точность*
Спецификация вывода: ***значение суммы***

- *параметром рекурсии* данного типа циклического процесса будет *слагаемое*, так как его значение монотонно убывает с добавлением каждого нового слагаемого.
- *тривиальным случаем*, очевидно будет случай, когда *абсолютная величина слагаемого* окажется *меньше* значения *точности* просчетов. В этом случае естественно взять за значение суммы нуль.
- на этапе декомпозиции задачи, естественно к переданному в алгоритм значению слагаемого добавить сумму оставшихся слагаемых. При этом первое из оставшихся слагаемых будет связано с переданным данному алгоритму через фактический параметр слагаемым рекуррентной формулой:

слагаемое:=слагаемое/(i + 1), где

i – номер введённого слагаемого.

```
| Задача 17
|     Построить рекурсивный алгоритм по просчету суммы вида
|     S = 1 + 1/2! + 1/3! + ...
| с точностью "точность".

алг вещ значениеСуммы(арг вещ слагаемое, точность, цел i)

abs(слагаемое) < точность
знач := 0
знач := слагаемое + значениеСуммы(слагаемое/(i + 1), точность, i + 1)
```

Рис. 8. 30 Рекурсивная версия реализации КМВ-цикла

- В таком случае обращение к записанному алгоритму будет выглядеть так:

  ……
  вывод «Сумма =», значениеСуммы(1, 0.0001, 1)
  ……

Общий формуляр алгоритма таков:

	A	B	C
1	значениеСуммы =	?	
2	слагаемое =	1	
3	точность =	0.6	
4	i =	1	

Рис. 8. 31 Вид общего формуляра

<u>Реализация аналога И – цикла</u>

Задача 18 Реализовать рекурсивный алгоритм общего решения уравнения *f(x) = 0* методом половинного деления с контролем за окончанием просчётов по малости невязки.

Спецификация ввода: *a b точность*
Спецификация вывода: *значение решения*

- *параметр рекурсии* для данного типа циклического процесса будет задаваться опосредовано, за счёт указания границ отрезка, содержащего корень. Дело в том, что зная границы отрезка, а для других итерационных методов – приближённое решение, можно определить невязку.
- *тривиальный случай* определяется такими значениями границ, *средняя точка* которых формирует *невязку, меньшую* заданной *точности*. Следовательно, это точка может быть взята за решение уравнения с заданной точностью.
- *декомпозиция* определяется необходимостью повторного просчёта (повторной бисекции отрезка) при новых значениях граничных точек.

Эти точки должны быть такими, чтобы между ними по-прежнему содержался корень уравнения.

- Обращение к рекурсивному алгоритму, записанному ниже, производится естественным образом.

```
| Задача 18
|     Реализовать рекурсивный алгоритм общего решения
| уравнения f(x) = 0 методом половинного деления
| с контролем за окончанием просчётов по малости невязки.
|
|       Спецификация ввода    :
|             a b точность
|       Спецификация вывода :
|             значение решения

алг вещ методПоловинногоДеления(арг вещ a, b, точность)
```

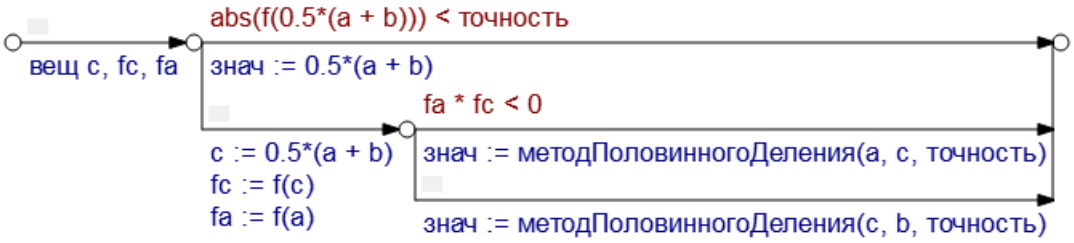


Рис. 8. 32 Рекурсивная версия реализации И-цикла

<u>Реализация обработки линейной таблицы (задача анализа).</u>

Задача 19 Составить рекурсивный алгоритм, который в линейной табличной величине элементов целого типа подсчитает количество нулей.

Спецификация ввода: *элементы вектора*
Спецификация вывода: *количество нулевых элементов*

- в качестве *параметра рекурсии* при обработке табличных величин чаще всего выступает индекс обрабатываемого элемента таблицы. Заметим, что от входящих параметров заголовка, описывающих вектор, можно было бы отказаться, если бы таблица была описана как глобальная величина.
- *тривиальный случай* определяется значением индекса, превосходящим размерность таблицы. Естественно положить, что количество нулей в этой части вектора равно нулю.
- *декомпозиция* определяется необходимостью нового анализа очередного элемента вектора на равенство 0.

```
| Задача 19
|    Составить рекурсивный алгоритм, который
| в линейной табличной величине элементов целого типа,
| подсчитает количество нулей.

алг цел количествоНулей(арг цел n, поз, арг цел таб вектор[1 : n])

поз > n  /* Указатель - за пределами индексного поля */
знач := 0
      вектор[поз] = 0
      знач := количествоНулей(n, поз + 1, вектор) + 1
знач := количествоНулей(n, поз + 1, вектор)
```

Рис. 8. 33 Рекурсивная версия обработки линейной таблицы (анализ)

Задача 20 Составить рекурсивный алгоритм просчёта скалярного произведения двух линейных таблиц (векторов).

- При решении данной задачи *параметр рекурсии, тривиальный случай* и *декомпозиция задачи* производятся совершенно аналогично тому, как это делалось при решении задачи 19. Поэтому сразу приведём текст алгоритма:

```
| Задача 20
|    Составить рекурсивный алгоритм просчёта
| скалярного произведения двух линейных таблиц (векторов)

алг вещ скалярноеПроизведение(арг цел n, инд, арг цел таб ВекторА[1: n], ВекторВ[1: n])

инд > n /* Индекс - за пределами индексных значений векторов*/
знач := 0
знач := ВекторА[инд]*ВекторВ[инд] + скалярноеПроизведение(n, инд + 1, ВекторА, ВекторВ)
```

Рис. 8. 34 Рекурсивная версия реализации просчета скалярного произведения векторов

Реализация обработки строковых (литерных) величин (задача преобразования).

Задача 21 Составить рекурсивный алгоритм построения литерной величины, записанный теми же символами, что и данная, но в обратном порядке.

- Вообще говоря, данную задачу можно решать в двух вариантах: во-первых, на основе исходного текста построить новый, но записанный «задом наперёд»; во-вторых, преобразовать сам предлагаемый текст.
  - в качестве *параметра рекурсии* при обработке литерных величин чаще всего выступает индекс – указатель на обрабатываемый символ текста. В данной постановке задачи «обращение» особое внимание следует обратить на синхронность обмена символами, равноудалёнными от концов текста. Внимание! В силу того, что за один шаг преобразований обмену будут подвергаться два символа (!) – количество повторов будет наполовину меньше. Что произойдёт, если об этом обстоятельстве забыть?
  - *тривиальный случай* определяется значением индекса, превосходящим середину длины текста. Естественно положить, что в этом случае текст никакому изменению подвергаться не должен (но только в случае, если производится обращение без привлечения дополнительных величин). А если это не так?
  - *декомпозиция* определяется необходимостью обмена значениями новой пары равноудалённых элементов текста.

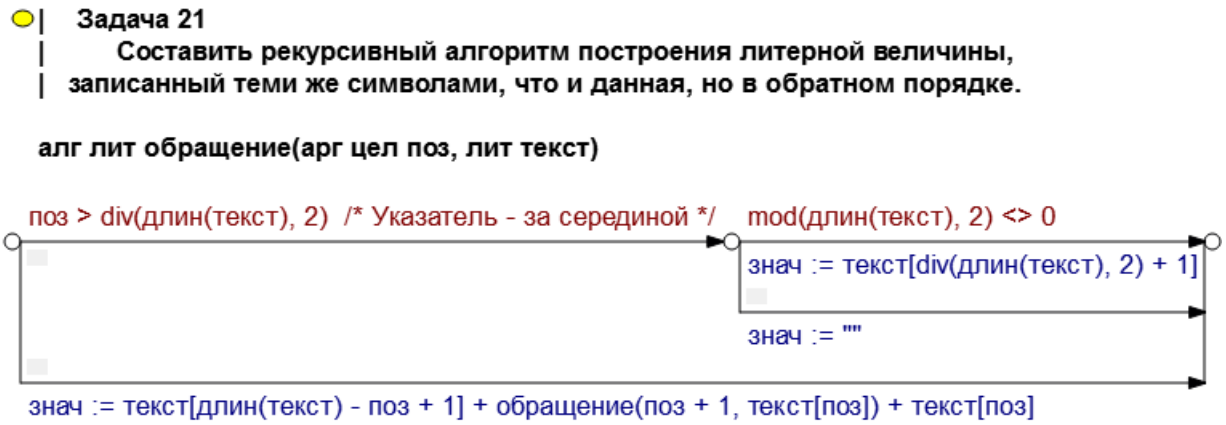


Рис. 8. 35 Рекурсивная версия реализации обращения строковой величины

```
алг лит обращение(арг цел поз, лит текст)
нач
    если(поз > div(длин(текст), 2)  /* Указатель - за серединой */)
      то
        если(mod(длин(текст), 2) <> 0)
          то
            знач := текст[div(длин(текст), 2) + 1]
          иначе
            знач := ""
        все
      иначе
        знач := текст[длин(текст) - поз + 1] + обращение(поз + 1, текст[поз]) + текст[поз]
    все
кон
```

Задача 21.1

Составить рекурсивный алгоритм построения литерной величины, записанный теми же символами, что и данная, но в обратном порядке, не привлекая других строковых величин (КУМИР, Delphi)

```
алг лит самообращение(арг цел поз, арг рез лит текст)
нач
    поз <= div(длин(текст), 2)  /* Указатель не дошел до середины */
    сим Буква
    Буква := текст[длин(текст) - поз +1]
    текст[длин(текст) - поз +1] := текст[поз]
    текст[поз] := Буква
    самообращение(поз + 1, текст)
кон
```

Рис. 8. 36 Рекурсивное обращение строковой величины без привлечения дополнительной переменной

К оглавлению К следующему параграфу К предыдущему параграфу

§9. О СТРУКТУРНОСТИ В ПРОГРАММИРОВАНИИ[24]

9.1 Совершенствование методов программирования:

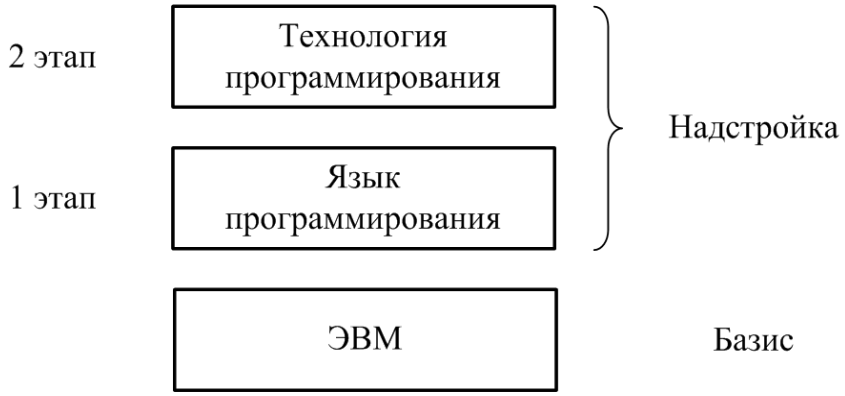


Рис. 9. 1 Иерархия средств описания алгоритмов для ЭВМ

9.2 Наиболее популярные технологии

- структурная технология;
- модульная технология;
- нисходящая технология или технология программирования «сверху вниз»;
- восходящая технология или технология программирования «снизу вверх»;
- R-технологи (отечественная разработка ИК АН УССР);
- объектно-ориентированная технология.

Общая особенность – это *структурность*.

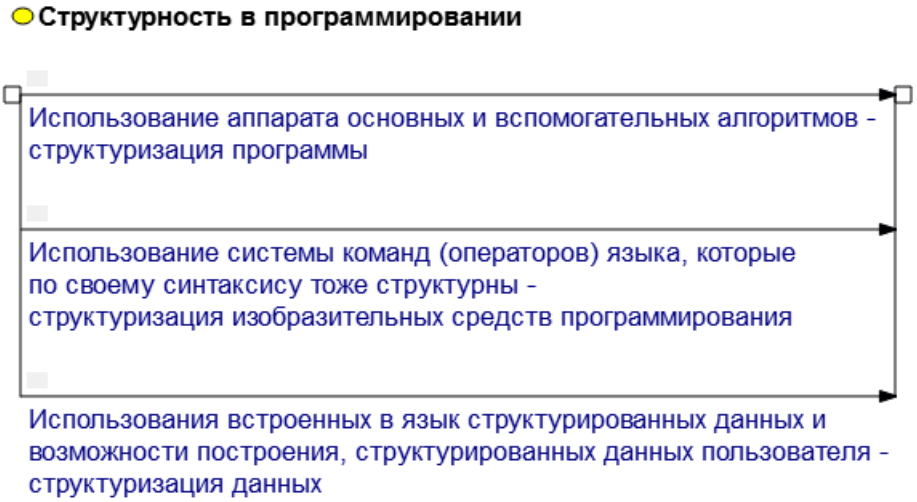


Рис. 9. 2 Классификация понятия «структурность»

[24] Ознакомиться с иллюстрирующими Р-схемами данного параграфа можно в документе 05 <Док. 05>

- на смену *программированию в машинных командах (кодах)* пришло программирование на *языках высокого уровня*, как более производительное в смысле написания кода и надёжное – в смысле качества программ.
- Результат: управляющие структуры (операторы, команды) языков высокого уровня строго определенным образом отображаются, проецируются трансляторами или интерпретаторами языка на структуру машинных команд.
- *структуры данных* отображаются на *структуру оперативной памяти* (ОЗУ) компьютера (память компьютера *дискретна*).

▪ Рассмотрим вопрос о том, каким образом данные хранятся в самой ЭВМ, т.е. рассмотрим структуры представления данных в памяти ЭВМ.

9.3 Структура представления данных состоит из двух частей:

- *структуры хранения значения* объекта и
- *базовых алгоритмов, реализующих базовые операции* над этими значениями.

▪ Таким образом:

Представление данных= Структура хранения +
Реализация базовых операций

- *Структура хранения – это способ размещения, кодирования значений данного типа* в памяти ЭВМ.
- Вся память ЭВМ делится на две категории: *внутреннюю* и *внешнюю*.
- *Внутренняя* или оперативная память (ОП) предназначена для хранения программ и данных во время вычислений.
- *Внешняя память* (диски, ленты) служит для долговременного хранения информации.
- ОП состоит из множества единиц, имеющих адреса; или ОП представляет собой линейную последовательность минимальных адресуемых единиц.

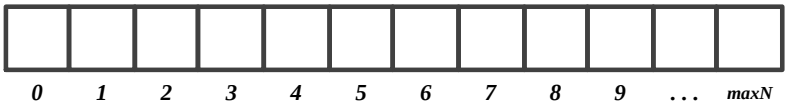


Рис. 9. 3 Структура оперативной памяти ЭВМ

- В современных компьютерах такими единицами являются *байты*, состоящие из *8 битов*: каждый бит может принимать два значения: 0 и 1, т.е. в байте может содержаться 256 различных значений.

- *По адресу* можно обратиться и к *двум последовательно расположенным байтам – машинному слову* или просто слову, и к *двойному слову*. В качестве минимально адресуемой единицы стали выбирать такой объем памяти, в котором можно разместить один символ машинного алфавита: для персональных ЭВМ (ПЭВМ) это и есть *байт*.
- *Каждая единица ОП имеет свой адрес*, т.е. порядковый номер; нумерация адресов начинается с нуля.

9.4 Распределение памяти

Любая программа существует в двух вариантах:

- как *текст на некотором языке программирования* и
- как *последовательность машинных команд*.
- С началом работы программы, т.е. с началом ее загрузки в память компьютера, каждый ее объект (константа, переменная, элемент структуры и т.д.) связывается с некоторым объектом самой ЭВМ.
    - Объект имеет собственное имя, которое обычно называют указателем (адресом). Эта связь устанавливается в момент создания объекта, т.е. в момент начала его существования. Этот процесс называется *распределением памяти*.
    - В высокоуровневых языках программирования со строгой типизацией этот же процесс носит название *описания переменных*.
- В современной терминологии *для хранения значений простых переменных* используются *байты*, *слова* и *двойные слова*. С какой по объёму ячейкой памяти (байтом, словом или двойным словом) будет связана переменная, зависит от *типа переменной*.
- Связь «*переменная – адрес*» можно проиллюстрировать так:

Рис. 9. 4 Иллюстрация связи «простая переменная – адрес»

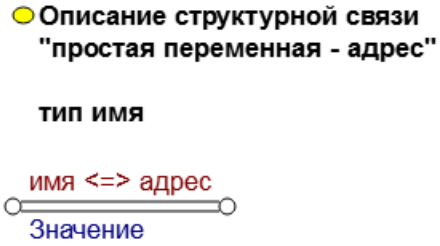


Рис. 9. 5 Иллюстрация связи «простая переменная – адрес»

- При отображении на машинную память стандартных (предоставляемых самим языком программирования) структурных данных происходит несколько иная картина.

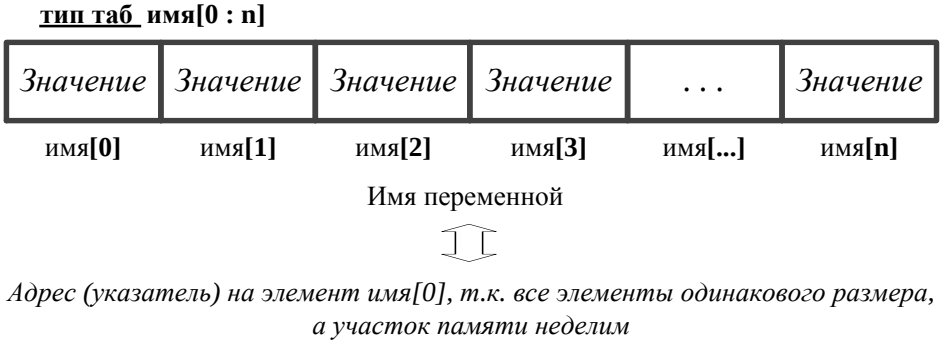


Рис. 9. 6 Иллюстрация связи «линейный массив – адрес»

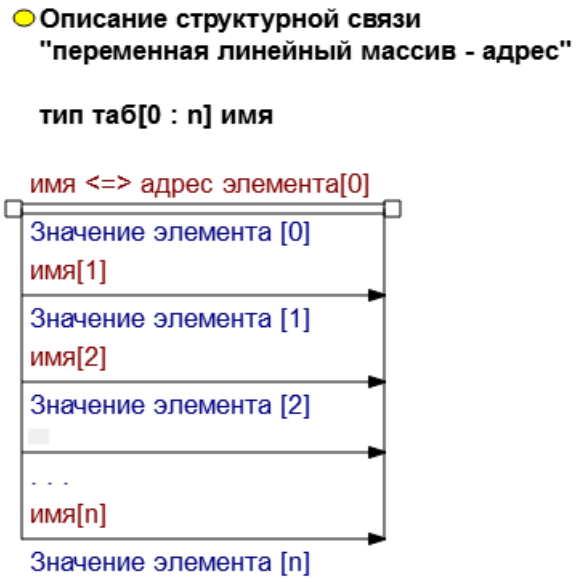


Рис. 9. 7 Иллюстрация связи «линейный массив – адрес»

- Для прямоугольных табличных величин размещение элементов, опять же, в неделимом участке памяти, производится следующим образом:

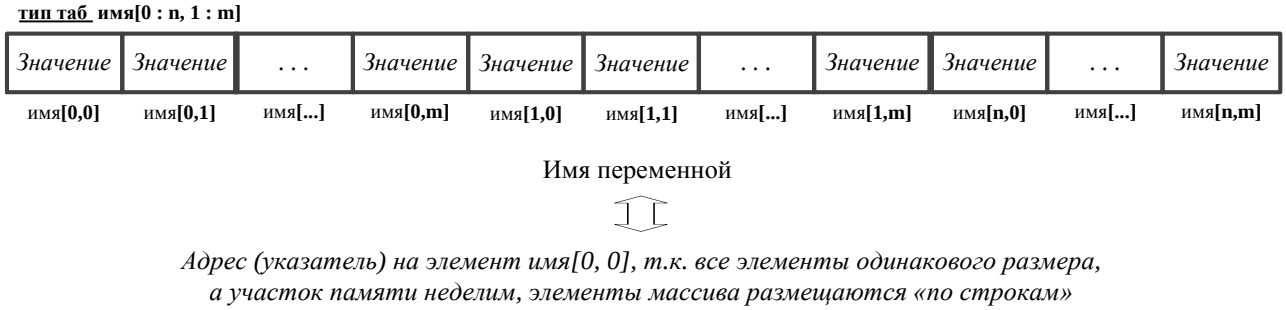


Рис. 9. 8 Иллюстрация связи «прямоугольный массив – адрес»

Рис. 9. 9 Иллюстрация связи «прямоугольный массив – адрес»

- На *машинном уровне* обращение к элементам табличной величины происходит за счет указания имени этой величины и индекса (или набора индексов) и реализуется так:

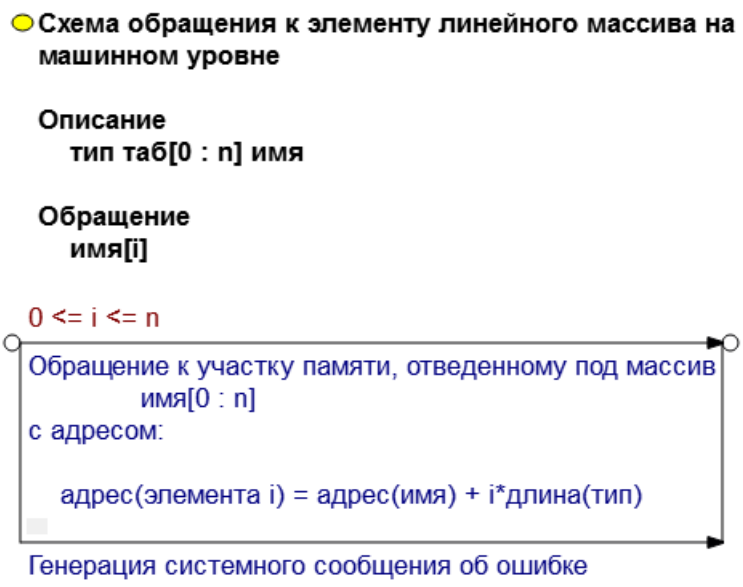


Рис. 9. 10 Алгоритм извлечения значения элемента линейного массива

- Для линейной таблицы понятие *приведённого индекса* совпадает с понятием индекса, поэтому алгоритм извлечения значения его элемента будет простым:

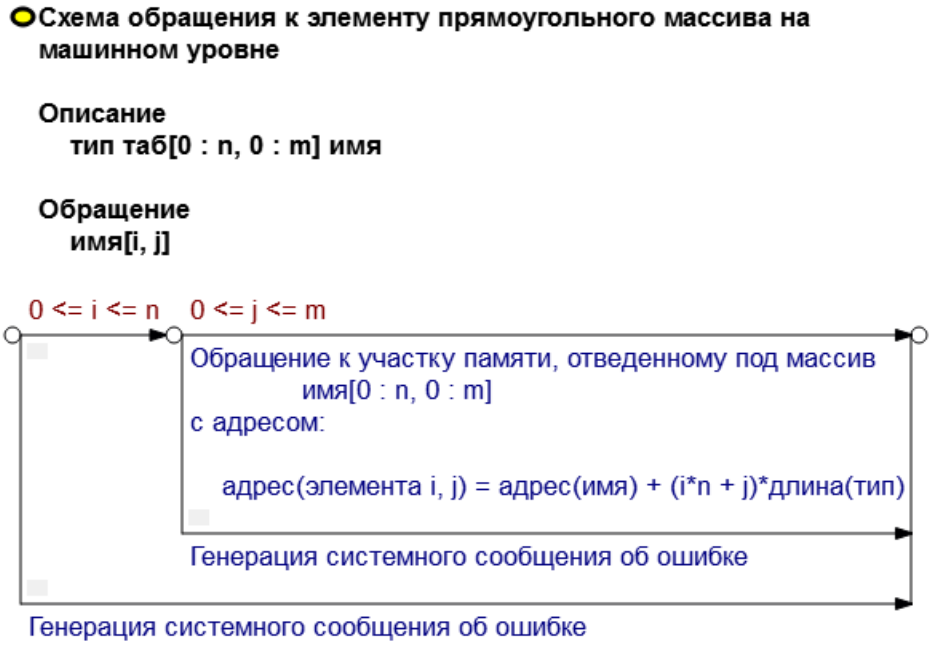


Рис. 9. 11 Алгоритм извлечения значения элемента прямоугольного массива

- Для случая же переменных *записного типа* отображение компонент значения на фрагменты машинной памяти производится по схеме, сходной с установлением соответствия для линейных табличных величин:

имя : **запись**
поле1 : **тип1**;
поле2 : **тип2**;
. . .
полек : **типk**
кц записи

тип1	**тип2**	, , .	**типk**
Значение	*Значение*	*Значение*	*Значение*
*поле**1***	*поле**2***	**. . .**	*поле**k***

Имя переменной.*полеТ*

Адрес (указатель) на элемент имя.полеТ, т.к. размер элемента записи определяется размером типа поля, но участок памяти неделим

Рис. 9. 12 Иллюстрация связи «поле записи – адрес»

Описание структурной связи "переменная типа запись - адрес"

имя : запись
поле1 : тип1;
поле2 : тип2;
. . . ;
полек : типk;
кц записи

имя.поле = поле1
Значение элемента поля 1
имя.поле = поле2
Значение элемента поля 2
. . .
. . .
имя.поле = полек
Значение элемента поля k

Рис. 9. 13 Иллюстрация связи «поле записи – адрес»

9.5 Способы распределения памяти

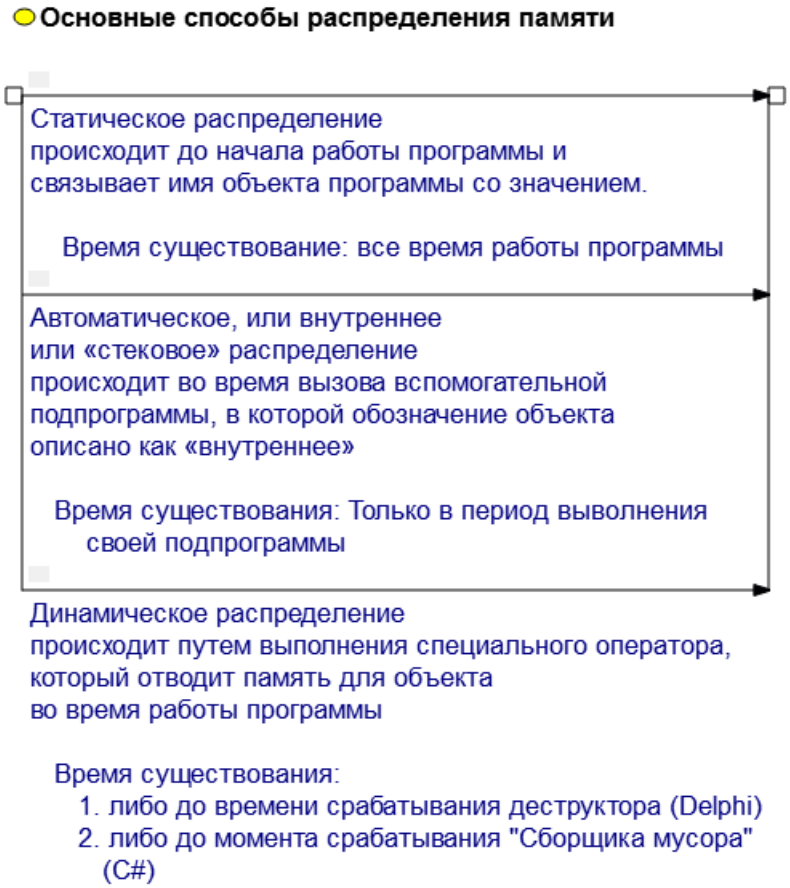


Рис. 9. 14 Моменты выполнения основных способов распределения памяти

- Обычно вся память, отведенная для программы, состоит из следующих частей:

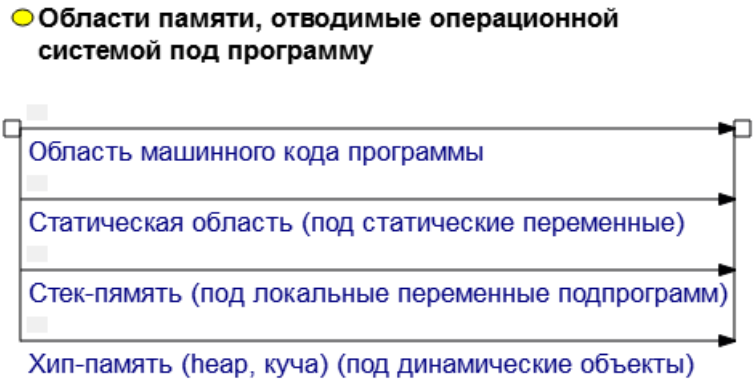


Рис. 9. 15 Программные области памяти

- *Указатели* (имена) *статических объектов* (ввиду их постоянства) встраиваются транслятором прямо в машинные команды.
- Указатели *стековых* и *динамических объектов* являются значениями соответствующих переменных типа «указатель», и в этих случаях их обозначения связывается с указателями не напрямую, как у статических

объектов, а косвенно – с указателями на переменную типа «указатель», значение которой уже является указателем на сам объект.

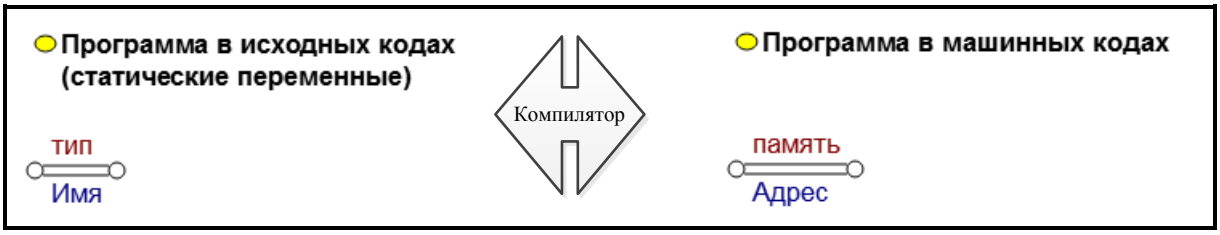


Рис. 9. 16 Встраивание адресов статических величин

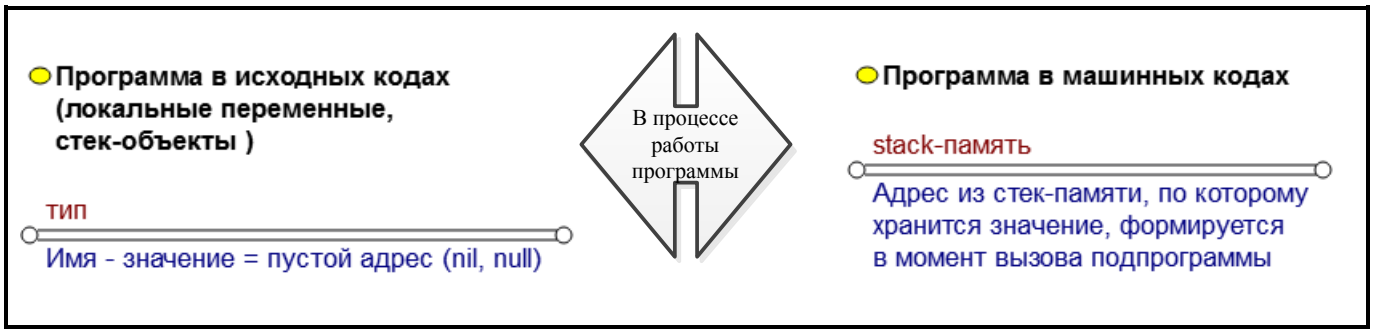


Рис. 9. 17 Встраивание адресов локальных и стек-переменных

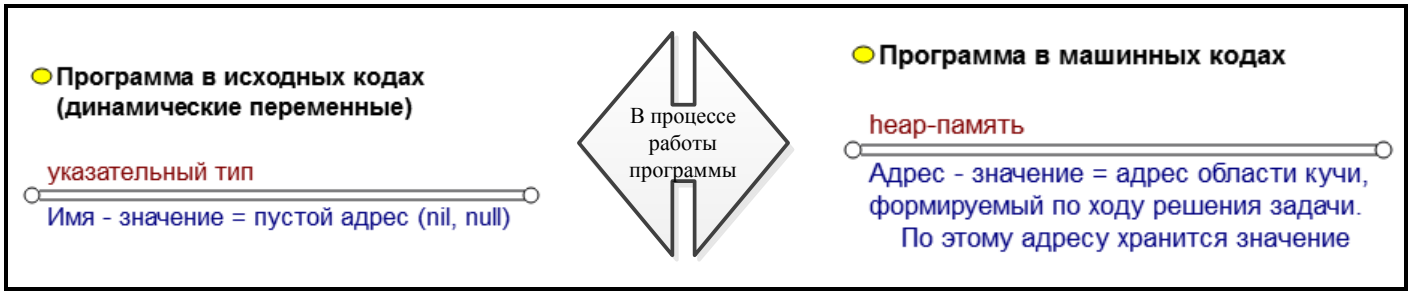


Рис. 9. 18 Встраивание адресов динамических (ссылочных) переменных

- Графически основные моменты формирования значения динамического объекта выглядят следующим образом:

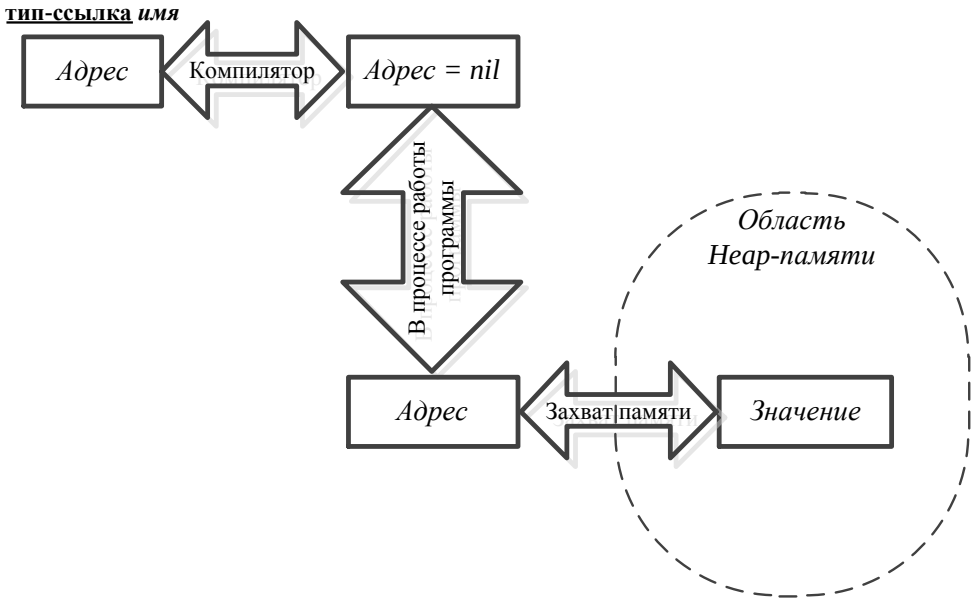


Рис. 9. 19 Иллюстрация формирования содержимого динамического объекта

9.6 Виды распределения памяти

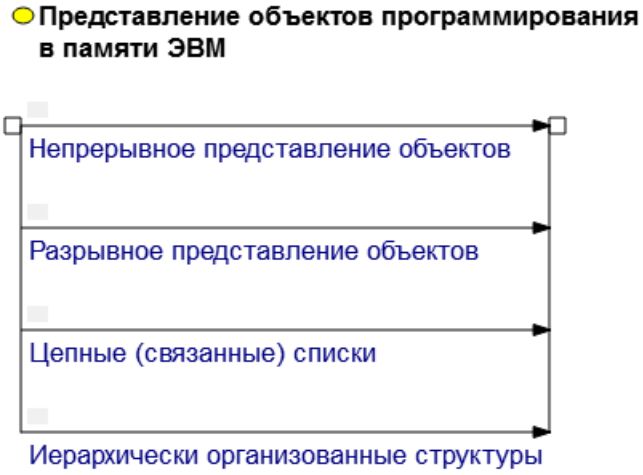


Рис. 9. 20 Способы представления объектов в памяти ЭВМ

9.6.1 Непрерывное представление объектов в памяти

- При непрерывном представлении для размещения всего значения объекта отводится *непрерывная последовательность байтов – область*.
- Если объект имеет *простой тип*, то в области размещается двоичный код всего значения как единого целого (например, числа – целые и вещественные, символы, указатели).
- В современных компьютерах для простых типов область может состоять из 1, 2, 4, 8, 16, 32 байтов, при этом *база области* (адрес ее начала) обычно кратна ее длине. На рис.9.21 показан пример размещения объекта в ОП, область для которого равна двум байтам, а сам объект размещен в байтах с номерами 4 и 5 (база равна 4).

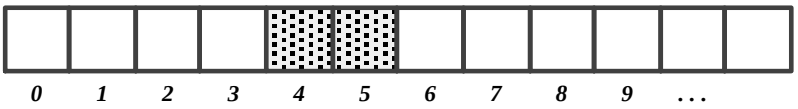


Рис. 9. 21 Структура компоненты при непрерывном представлении

- Если объект имеет структурный тип, т.е. его значение состоит из компонент, тогда внутри области от ее начала подряд выделяются подобласти-поля для значений компонент. Их нумеруют слева направо, считая, что номер самого левого поля равен нулю.
- В общем случае требуется знать следующие величины, совокупность которых можно назвать паспортом области:
  - *база области* ***a***, т.е. адрес ее первого байта;
  - длины полей области $L_0, L_1, \ldots$;
  - *число* полей.
- Начало ***i***-го поля в области обычно характеризуется *не адресом в общей памяти* (абсолютным адресом), а *относительным адресом*, или смещением, $S_i = a_i - a$, где a_i – абсолютный адрес поля, а ***a*** – *база области*. Смещение вычисляется по длинам полей следующим образом:

$$S_0 = a_0, \qquad S_i = L_0 + L_1 + \ldots + L_{i-1}.$$

- Частным случаем области с полями является *вектор (линейный массив, таблица)*, в котором все поля имеют одну и ту же длину ***L.*** Для вектора смещение ***i***-го поля есть

$$S_i = (i - 1) * L.$$

9.6.2 Разрывное представление объектов в памяти

- При *разрывном* (*ссылочном*, *списковом*, *связанном*) представлении для каждой части структурного значения (компоненты) отводится своя область, и эти области могут занимать произвольные места в памяти. Они называются элементами и представляют собой области, по крайней мере, с двумя полями (рис. 9.22), причем в одном или нескольких полях хранятся указатели на другие элементы.

узел : **запись**
    содержимое : **тип***;*
    адресная Часть
кц записи

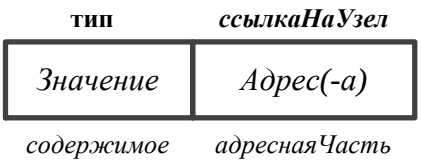


Рис. 9. 22 Структура компоненты при разрывном представлении

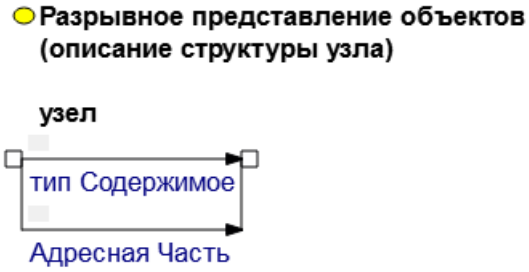


Рис. 9. 23 Общая структура узла

9.6.2 Списочное представление объектов в памяти

- Цепные списки представляют собой *разрывно* для их *компонент представленную структуру*, в которой каждый элемент (кроме информационных) имеет *одно* или *два* поля с указателем.
- В практике программирования используются следующие виды цепных списков: *линейные* и *циклические*, *одно*- и *двунаправленные*.
  - В линейных списках все элементы линейно упорядочены за счет своей организации (особенностей построения).

Рис. 9. 24 Особенности организации однонаправленного списка

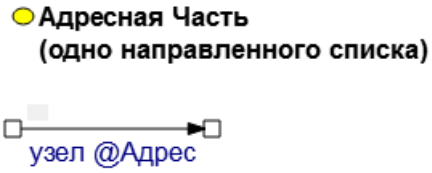


Рис. 9. 25 Строение адресной части однонаправленного списка

Рис. 9. 26 Организация однонаправленного списка

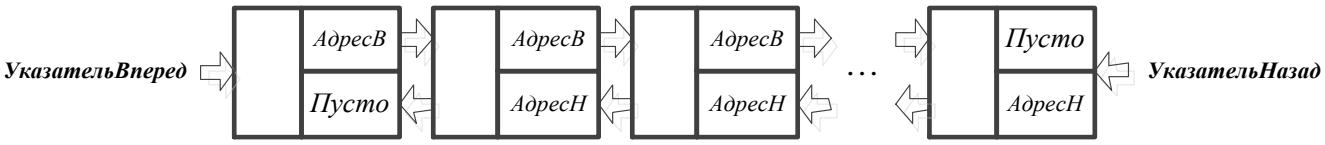


Рис. 9. 27 Особенности организации двунаправленного списка

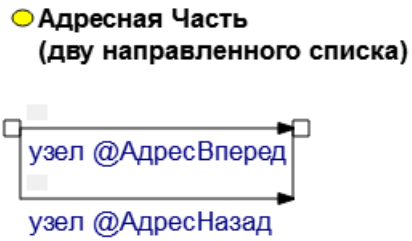


Рис. 9. 28 Строение адресной части двунаправленного списка

Рис. 9. 29 Организация двунаправленного списка

- Циклический список отличается от линейного тем, что последний элемент содержит не пустой указатель *пусто* (***nil***), а указатель на голову списка: при этом головным (условно первым) элементом может быть любой элемент.

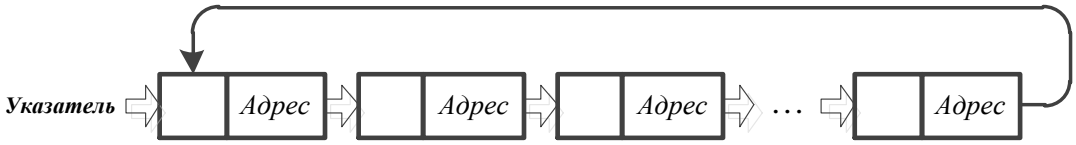


Рис. 9. 30 Особенности организации однонаправленного циклического списка

Рис. 9. 31 Организация однонаправленного циклического списка

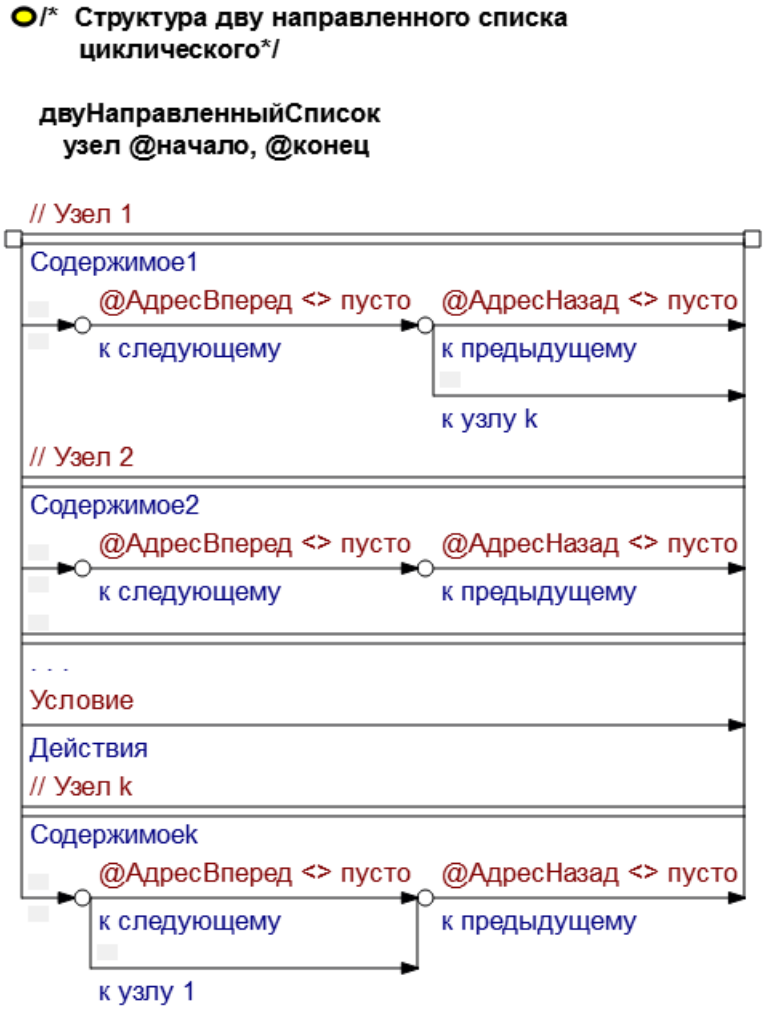


Рис. 9. 32 Организация двунаправленного циклического списка

9.6.3 Иерархически организованные объекты

- Иерархически организованные объекты представляют собой *разрывно* для их *компонент представленную структуру*, в которой каждый элемент (кроме информационных) имеет не менее *двух* полей с указателями.
- Такие структуры еще называют *сложно организованными*.
- В практике программирования используются следующие виды сложно организованных структур – *бинарные деревья*, *деревья общего вида* и *графы*.

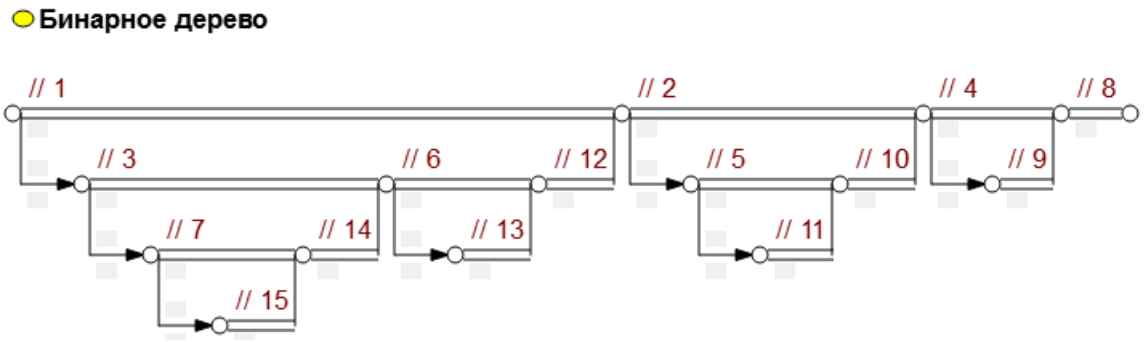


Рис. 9. 33 Р-схема связи узлов полного бинарного дерева глубиной 3

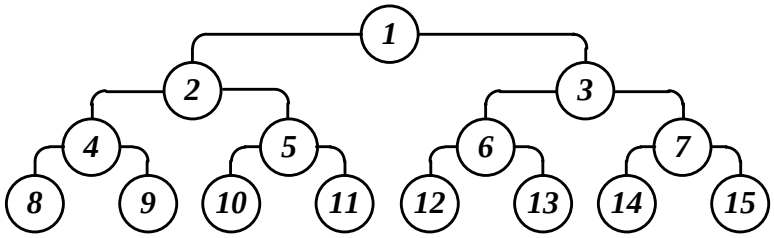


Рис. 9. 34 Схема связи узлов полного бинарного дерева глубиной 3

9.6.4 Структура внешней памяти

- Основная структура данных, с которой чаще всего приходится встречаться в ходе решения практических задач (или *структура представления внешней памяти* ЭВМ) – это *файл*.
- *Файл* это именованная область данных на внешнем носителе информации.
- Строгое определение: *файл – это набор логически связанных между собой информационных объектов, созданный для решения определенной задачи*. Чтобы подчеркнуть физическую (а не логическую) интерпретацию этого понятия, иногда вместо слова «файл» говорят «набор данных».
  - Обычные операции с файлами – *добавление*, *удаление* и *обновление* компонент, а также *подготовка к открытию* файла и его *закрытие*.
- Наиболее простой тип файлов – это файл с *последовательным доступом*: в каждый момент доступна лишь одна компонента последовательности, которая определяется текущей позицией механизма доступа.

- Текущая позиция (указатель на компоненту файла) может меняться, определяя либо следующую компоненту, либо первую компоненту всей последовательности (всего файла).
- Таким образом, файл делится на две части с помощью указателя: на *просмотренную часть файла* и *не рассмотренную часть файла.*
- Физическими носителями информации для файлов являются диски, магнитные ленты, флешки и другие магнитные и оптические носители информации. На них файлы хранятся с особыми пометками, обозначающими название файла и метку его начала, а также метку конца файла. Благодаря их наличию, операциями
  - начало просмотра (типа Reset) указатель файла устанавливается на метку начала (перед первой компонентой файла);
  - определения конца файла (типа EOF) определяется возможность прочтения очередной компоненты файла.

9.7 Структура данных ООП

- Получаемая с использованием принципов объектно-ориентированного программирования информационная модель реальности представляет собой

Модели данных + Модели связей

- Подобная конструкция носит название *структуры данных* или (в современной терминологии) *Абстрактные Типы Данных* (АТД), или (в терминах объектно-ориентированного программирования) *объектные типы* (*классы*).
- Позже в настоящем курсе будут введены построенные автором и широко используемые в практике программирования структуры данных или *АТД* и их реализации различными способами, учитывающими разные возможности разных языков программирования.
- Основу наиболее популярных и отвечающих современным требованиям базовых курсов программирования составляют такие понятия, как
  - понятие *класса* или *объектного типа*, или в школьной русской терминологии *исполнителя*;
  - понятие о нисходящей разработке алгоритма и
  - понятие структуры данных или АТД.
- Поэтому далее переходим к описанию сути *класса* (*исполнителя*) и *нисходящей разработки*.

К оглавлению К приложению К предыдущему параграфу

§10. Основная задача программирования. Понятие исполнителя

10.1 Определения и соглашения:

Определение. *Исполнитель (И) (класс)* – это человек, организация, механическое или электронное устройство, робот и т.д., умеющий выполнять определенный набор действий в определенной среде.

- Каждое действие, выполняемое ***И***, носит название *предписания*.
- Каждый *реальный* ***И*** имеет свою *систему предписаний*.
- В данном курсе речь идет исключительно об ***И***, предназначенных для *обработки информации*.
- Действия, производимые информационным ***И*** над объектами (величинами), приводят к *изменению их состояний* (значений).
- Объекты, обрабатываемые ***И***, могут быть *внутренними* и *внешними*.
  - *Внутренние* объекты ***И*** называются еще его глобальными величинами, так как они относятся ко всем предписаниям данного ***И***, то есть ко всему ***И*** в целом.
  - *Внешние* объекты ***И***, называемые еще *локальными* величинами или *параметрами* предписания, предназначены для связи с ***И*** при вызове его конкретного предписания.

```
исп имяИсполнителя
  используются <Список экспортируемых исполнителей>
  раздел описаний
    конст < Список констант исполнителя>
    тип   < Список типов исполнителя>
    перем < Список переменных исполнителя>
    < Список объявления методов исполнителя>
  раздел реализаций
    < Реализация методов, объявленных
      в разделе описаний исполнителя>
кц исп имяИсполнителя
```

Рис. 9. 35 Общая схема оформления исполнителя

- *Внешние переменные* или *объекты* обозначаются как *вид величины* словами **арг**, **арг рез**;
- *Внутренние переменные* описываются внутри описанного исполнителя, в разделе константы, как промежуточные величины. Но обращаться к ним может любой внутренний алгоритм данного исполнителя.

10.2 Каково назначение аппарата И?

- Решение серьезных задач приводит к необходимости *разбиения* их *на подзадачи*.
- В этом случае считают, что *подзадачи* уже *решены*, т.е. составлены реализующие их решение подпрограммы.
- В таком случае, при решении *головной* задачи можно пользоваться полученной системой вспомогательных алгоритмов-предписаний.
  - Каждая подзадача, в свою очередь, может быть для своего решения разбита на *подзадачи второго уровня*, решение которых тоже образует своеобразную систему предписаний, которой может пользоваться как любое предписание (вспомогательный алгоритм) соответствующего основного алгоритма второго уровня, так и головной алгоритм предыдущего первого уровня.
  - Этот процесс выделения подзадач новых уровней может быть продолжен в случае необходимости и дальше.
- Так как вызов программы – это не что иное, как приказ на выполнение определенных действий (предписаний), то программу с группой вспомогательных подпрограмм можно считать ***И*** с некоторой системой предписаний.
- Про саму программу можно в этом случае говорить, что она реализована группой программ на базе таких-то ***И***, где каждая программа любого уровня реализует некоторое предписание.

- Изображая описанное выше в виде обобщенной схемы, получаем

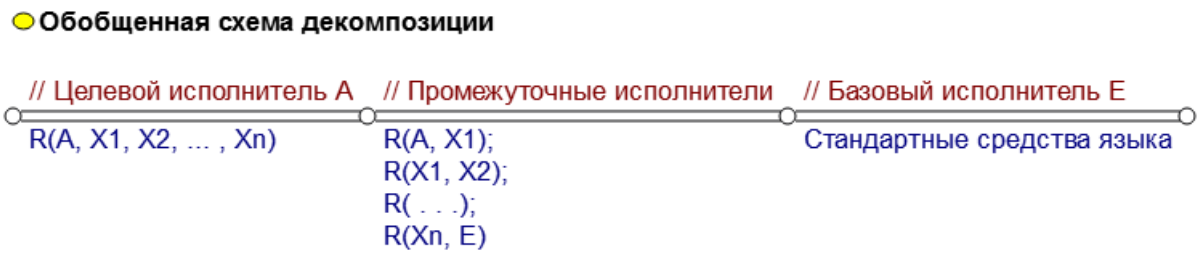


Рис. 9. 36 Общая Р-схема процесса декомпозиции

> ***Основная задача программирования*** состоит в следующем: по *внешнему* описанию *искомого исполнителя* ***A***, который необходимо создать, и по *внешнему описанию базового исполнителя* ***E***, который считается существующим, надо построить представление ***R(A, E)*** (*текст на некотором языке программирования*), которое описывает работу *исполнителя* ***A*** в терминах (предписаниях) ***E*** и позволяет реализовать ***A*** на основе ***E***.

Рис. 9. 37 Детализированные реализации исполнителей

где X_1, X_2, X_3, … , X_n – промежуточные исполнители, придуманные в процессе решения задачи. При этом текст ***R(A, E)*** представляет собой композицию реализаций вида:

$$R(A, E) = R(A, X_1) * R(X_1, X_2) * \ldots * R(X_n, E).$$

Определение 2 *Композиция представлений* – это объединение текстов $R(A, X_1)$, $R(X_1, X_2)$ и так далее в указанном порядке.

- Согласно технологии программирования «*сверху вниз*», создание текста ***R(A, E)*** состоит из последовательного применения операции

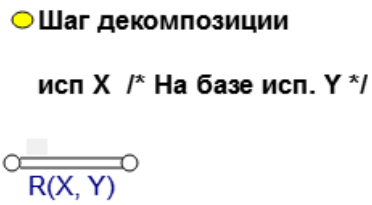


Рис. 9. 38 Описание шага декомпозиции

начиная с *внешнего описания исполнителя* ***A***, который надо построить, до тех пор, пока не дойдем до изначально данного (или реализованного) *исполнителя* ***E***.

- Эта операция называется *шагом декомпозиции* и является центральной (главной) в *нисходящей технологии программирования*. Сформулируем задачу шага декомпозиции.

> ***Задача шага декомпозиции*** состоит в том, чтобы по *внешнему описанию исполнителя* ***X*** построить *представление исполнителя* ***X*** на базе *введенного* (*гипотетического*, *придуманного* или *базового*) *исполнителя* ***Y*** и одновременно с этим дать *внешнее описание* самого используемого *исполнителя* ***Y***.

- Требования к проведению шага декомпозиции таковы:
  - *локальное*: по возможности большую часть работы переложить на нового *исполнителя* ***Y***;
  - *глобальное*: шаг декомпозиции должен приближать решение исходной задачи (реализацию *исполнителя* ***A***) к *базовому исполнителю* ***E***, то есть ***Y*** должен в некотором смысле быть «*ближе*» к ***E***, чем *исполнитель* ***X***.

Замечание 1. Обычно при выполнении шага декомпозиции *исполнитель* ***X*** реализуется на базе не одного *исполнителя* ***Y***, а нескольких *исполнителей* ($Y_1, Y_2, Y_3, \ldots, Y_n$).

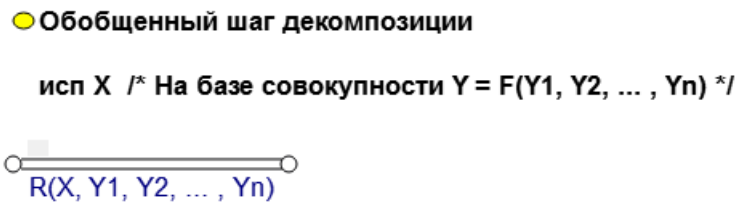


Рис. 9. 39 Описание обобщенного шага декомпозиции

Т.о., в формуле декомпозиции в общем случае

$X \to Y$ и $R(X, Y)$, где Y – это вектор из n компонент-исполнителей $Y = (Y_1, Y_2, Y_3, \ldots, Y_n)$.

- Заметим, что аналогичное замечание можно сделать и о базовом *исполнителе* ***E***.

Замечание 2. При выполнении шага декомпозиции разрешается использовать предписания исполнителей, внешние описания которых

  - либо были заданы, как *базовые*,
  - либо были ранее построены.

Замечание 3. Нередко при проведении новых шагов декомпозиции прежние (ранее реализованные) внешние описания исполнителей могут подвергаться уточнению, замене, исправлению и т.д., в связи с необходимостью получения более лаконичной или общей схемы решения задачи в целом. Т.о., программирование в соответствии с нисходящей технологией носит итерационный (уточняющий), а не линейный характер.

Замечание 4. Вообще говоря, технология нисходящего программирования (проектирования) включает в себя и средства контроля качества текстов $\boldsymbol{R(A, X_1)}$, $\boldsymbol{R(X_1, X_2)}$ и т.д., т.е. этапы *тестирования* и *отладки.* Причем затраты на эти этапы бывают ничуть не меньшими, чем затраты на создание указанных представлений $\boldsymbol{R(A, X_1)}$, $\boldsymbol{R(X_1, X_2)}$, … Данных вопросов базовый курс не затрагивает.

К оглавлению Ответы к заданиям К предыдущему параграфу

Приложение

- В представленных выше материалах мы смогли бегло коснуться основ разработки алгоритмов средствами Р-схем в среде редактора REditor.
- Возможно, в дальнейшем будет написано продолжение аналогичных разработок[25], ориентированных на решение более сложных задач, в частности, связанных с обработкой стандартных структур данных средствами таких языков программирования, как ***Delphi***, ***C++***, ***C#***, ***Java*** и т.п.
- Более интересными нами видятся аналогичные разработки, связанные с построением *Абстрактных Типов Данных* (***АТД***) и решением задач на их основе. В связи с этим встает вопрос об описании АТД в виде классов, а еще более ценно – в виде параметризованных классов.
- В заключение нам хотелось бы сделать следующие замечания (пояснения) на основе статей автора языка Р-схем И.В. Вельбицкого.

1. Истолкование сути (семантики) специальной дуги.

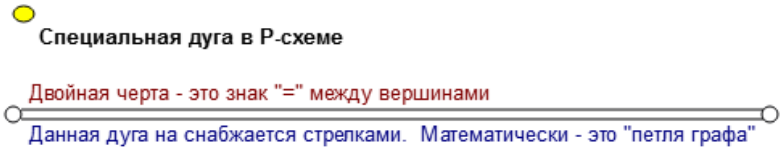


Рис. 1 Содержание (смысл) специальной дуги Р-схемы

- Таким образом, одно из назначений подобной дуги – «обозначать» цикл. Кстати, это было продемонстрировано в выше приведенных материалах.
- Кроме того, как было показано в п. 3.1 (рис. 3.3), специальная дуга может быть использована в тех случаях, когда над дугой Р-схемы вместо *Условия* надо записать ключевое слово. Этот прием мы будем использовать ниже, в иллюстративном примере описания *компонент класса*.

2. Истолкование сути (семантики) разных видов вершин графа (Р-схемы):

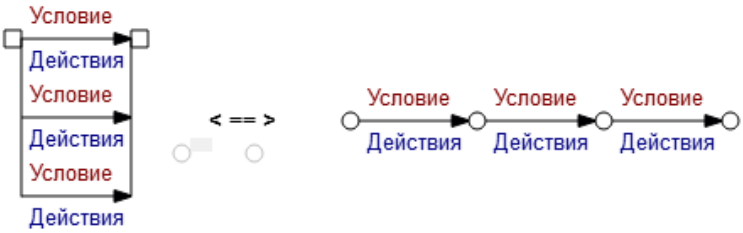


Рис. 2 Вершина квадратик Р-схемы

[25] Это зависит, в первую очередь, от той «отдачи», которую мы надеемся получить, внедряя в учебный процесс предлагаемые средства и, во-вторую, от временнóй составляющей, которой будет располагать преподаватель.

3. Вершина ***Квадратик*** предназначена для обозначения безальтернативной записи одиночных дуг. Р-схема, изображенная слева на рис. 2, эквивалентна последовательности дуг, изображенной справа на том же рисунке.

Рис. 3 Вершина ромбик Р-схемы

4. Вершина ***Ромбик*** предназначена для обозначения ожидания исполнения внешнего события, указанного в *Условиях* на дугах, исходящих из ромбика.

Рис. 4 Вершина прямоугольник Р-схемы

5. Вершина ***Вертикальный прямоугольник*** предназначена для обозначения параллельного выполнения дуг между прямоугольниками. На *Р-схеме* изображено параллельное выполнение трех дуг[26].

- Непосредственно ознакомиться с содержанием полной инструкции «ВТП Р_схемами Инструкция» можно в названном документе, любезно предоставленном нам И.В. Вельбицким [11 - 15] – научным руководителем сайта «Фонд Глушкова» (http://glushkov.org)
- В завершение приведем вариант оформления набора Р-схем при описании класса.
  - Во-первых, обратим внимание на возможность структуризации разработки алгоритма за счет использования ***Дерева проектов*** редактора *REditor*.

[26] Назначение остальных видов вершин пока особо не оговаривается. Возможно, оставляется для коммерческой версии *REditor* (Авт.).

- Однонаправленные АТД на C#
  - 01 База для построения АТД
    - 01.1 Класс узел однонаправленного спискаБезымянный
      - 01.1.1 Описание класса узел однонаправленного списка
      - 01.1.2 Методы класса узел однонаправленного списка
      - 01.1.3 Свойства класса узел однонаправленного списка
    - 01.2 Класс динамический вектор
  - 02 АТД Стек
    - 02.01 АТД Стек на базе класса узедОСписка
      - 02.01.1 Внешнее описание класса
      - 02.01.2 Реализация методов класса
      - 02.01.3 Реализация свойств класса
    - 03 АТД Очередь

Рис. 5 Дерево проектов разработки набора классов

01.1.1 Описание класса узел однонаправленного списка

```
/// <summary>
/// Класс для создания узлов однонаправленного списка узлов АТД
/// </summary>
/// <typeparam name="T">- тип нагрузки узла списка</typeparam>

public class узелОСписка<T>
```

// Поля данных	// Методы	// Свойства
T Содержимое;	public узелОСписка()	public T содержимое
узелОСписка<T> Следующий;	public узелОСписка(T нагрузка)	public bool естьСледующий
	public void связать(узелОСписка<T> Адрес)	public узелОСписка<T> следующий

```
/// <summary>
/// Структура-тип для организации связей в однонаправленных АТД
/// </summary>
/// <typeparam name="T">-тип нагрузки элементов моделируемой АТД</typeparam>

public struct одиночныйУказатель<T>
```

```
/// <summary>
/// Свойство по-чтению/по-записи получения/установления
    адреса очередного узла
/// </summary>

public T следующий
```

```
/// <summary>
/// Свойство-запрос наличия очередного узла моделируемой АТД
/// </summary>

public bool естьСледующий
```

T Следующий;

get
return Следующий;

set
Следующий = value;

get
return Следующий != null;

Рис. 6 Внешнее описание класса узелОСписок и структуры одиночныйУказатель<T>

01.1.2 Методы класса узел однонаправленного списка

```
/// <summary>
/// Конструктор создания указателя определенной структуры АТД
/// </summary>

public узелОСписка()

Содержимое = default(T);
Следующий = null;


/// <summary>
/// Конструктор создания нового узла структуры с данной нагрузкой
/// </summary>
/// <param name="нагрузка">- нагрузка создаваемого узла</param>

public узелОСписка(T нагрузка)

Содержимое = нагрузка;
Следующий = null;


/// <summary>
/// Метод установления ссылки ОТ текущего узла к узлу со ссылкой АДРЕС
/// </summary>
/// <param name="Адрес">- ссылка-адрес подключаемого узла</param>

public void связать(узелОСписка<T> Адрес)

Следующий = Адрес;
```

Рис. 7 Конкретизация методов класса

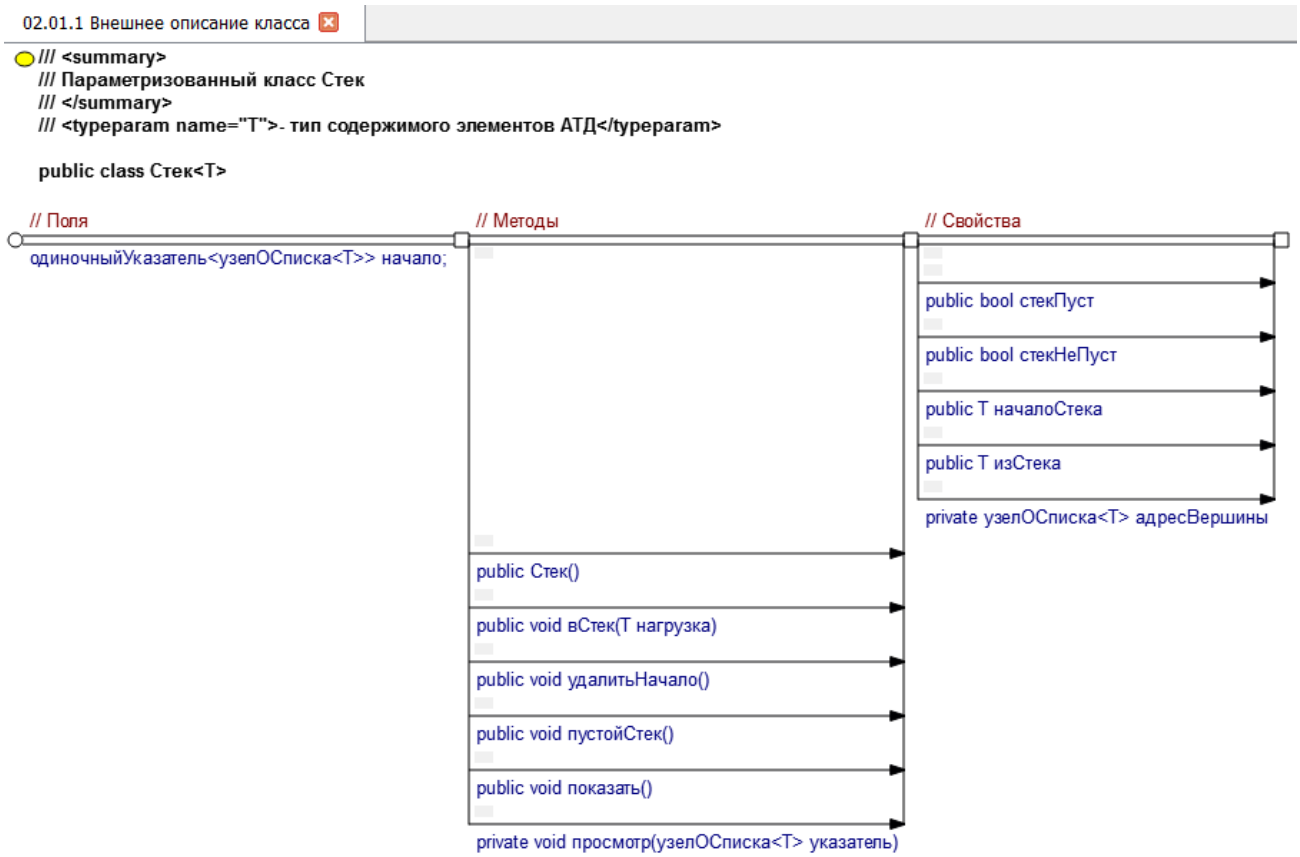


Рис. 8 Внешнее описание параметризованного класса Стек

К оглавлению Экпресс проверка теории К приложению

Ответы к заданиям

- Задание 1. Р-схема №1:

```
Р-схема: композиция структуры Цикл с предусловием
со структурой Ветвление
{
    while(Условие продолжения){
        Тело цикла
        if( Условие вложенного Ветвления){
            Действия альтернативы "то"
        }else{
            Действия альтернативы "иначе"
        }
    }
    if(Условие 1){
        Действия 1
    }else{
        Действие 2
    }
}
```

1. Выполняется цикл while, тело которого содержит «Тело цикла» и команду ветвления.
2. Внутри цикла сначала выполняются команды «Тела цикла», а затем…
3. В зависимости от значения истинности «Условия вложенного Ветвления» будут выполняться команды блока «Действия альтернативы «то»» или блока «Действия альтернативы «иначе»».
4. По выходу из цикла с предусловием, будет …
5. Просчитываться значение истинности «Условия1» и, в зависимости от полученного результата, будут выполняться …
6. «Действия альтернативы «то»» или блока «Действи1» или альтернативы «иначе» – блока «Действи2». <К постановке задачи>

- Задание 1. Р-схема №2:

```
Р-схема: композиция структуры Цикл с предусловием
со структурой Ветвление
{
    while(Условие продолжения){
        Тело цикла
        if( Условие вложенного Ветвления){
            Действия альтернативы "то"
            goto _l7e;
        }else{
            Действия альтернативы "иначе"
            goto _l7e;
        }
    }
    if(Условие 1){
        Действия 1
    }else{
        Действие 2
    }
    _l7e:;
}
```

1. Выполняется проверка условия вхождения в цикл while, тело которого содержит «Тело цикла» и команду ветвления.
2. Внутри цикла сначала выполняются команды «Тела цикла», а затем…
3. В зависимости от значения истинности «Условия вложенного Ветвления» будут выполняться команды блока «Действия альтернативы «то»» или блока «Действия альтернативы «иначе»».
4. Независимо от того, какая из альтернатив будет выполнена, по их завершении поток управления …
5. покинет тело цикла и перейдет к команде, которая имеет метку «_l7e». Т.о., в данном сочетании оператор while фактически выполняет роль «чистой» структуры Ветвления.
6. Если же поток управления не доберется до «Тела цикла», то будет выполнена структура Ветвления с «Условием 1», а по ее завершении оба потока сольются < К постановке задачи >.

К оглавлению *К литературе* *К ответам к заданиям*

ЭКСПРЕСС ПРОВЕРКА УСВОЕНИЯ НЕКОТОРЫХ ПОНЯТИЙ

- Для организации проверки необходимо обратиться к гиперссылке «Начать экспресс проверку».
- После загрузки среды тестирования необходимо:
  - Выбрать пункт меню «Тест»;
  - В подменю этого пункта выбрать пункт «Начать тест»;
  - В окнах появившейся формы «Начнем тест?!» указать по-русски свою Фамилию и Имя, а во втором окне – номер курса, и через пробел – первую букву специальности. После чего нажимаете кнопку «Ok».
  - После появившейся формы с информационным сообщением, нажимается кнопка «Ok».
  - С появлением вопросов теста, следует последовательно на них отвечать.
  - По завершению опроса (10 вопросов) выводятся диаграммы качества ответа и оценка.
  - Успешной Вам тренировки.

<Начать экспресс проверку>

Замечание: Далеко идущих выводов о качестве усвоения материала данного раздела авторы данных материалов вам не советуют делать. Реальное учебное тестирование на сайте snvtest.mf.brgu.by – значительно серьезнее. Поэтому авторы рекомендуют отнестись к изучению всего предложенного материала по-серьезному.

Успехов Вам в учебе.

К оглавлению *К экспресс проверке*

Литература

1. В.М. Глушков, И.В. Вельбицкий, Технология программирования и проблемы ее автоматизации, УСИМ, Киев, №6, 1976, p.75–93.
2. Вельбицкий И.В., Технология программирования / И.В. Вельбицкий, – Техника, Киев, Украина, 1984, 279 с.
3. Вельбицкий И.В. и др. Технологический комплекс производства программ на машинах ЕС ЭВМ и БЭСМ-6 / И.В. Вельбицкий, В.Н. Ходаковский, Л.И. Шолмов. – М.: Статистика, 1980. – 263 с.
4. INTERNATIONAL STANDART ISO/IEC 8631. Information technology-Program constructs and convention for their Representation – Second edition 1989.08.01 Geneve 20, Switzerland: ISO/IEC Copyright Office, p.7, 1989.
5. McHenry William K. R-Technology: A Soviet Visual Programming, Journal of Visual Languages and Computing, vol.1, #2, p.199-212, 1990.
6. I.V. Velbitskiy, Graphical Programming and Program Correctness Proof. (plenary report), 9-th IEEE COMPUTER SCIENCE & INFORMATION TECHNOLOGIES CONFERENCE, Armenia, Yerevan, 23-27 Sept.2013, IEEE Xplore, CSIT-2013.
7. Кнут Д. Искусство программирования на ЭВМ, т.1. Основные алгоритмы / Д. Кнут – М., Мир, 1976, 735 с.
8. Лингер Р., Теория и практика структурного программирования / Р. Лингер, Х. Миллс, Б. Уитт. – М., – Мир, 1982, 406 с.
9. Ахо А., Построение и анализ вычислительных алгоритмов / А. Ахо, Дж. Хопкрофт, Дж. Ульман. – М., Мир, 1979, 535 с.
10. Гудман С., Введение в разработку и анализ алгоритмов / С. Гудман, С. Хидетниеми – М., Мир, 1981, 366 с.
11. Электронный источник: Вельбицкий И.В., Пять основных преимуществ технологии программирования нового поколения, http://glushkov.org/wp-content/main%20advantages.pdf
12. Электронный источник: Вельбицкий И.В., Графическое программирование и доказательство правильности программ, http://glushkov.org/wp-content/131120CSITrus.pdf
13. Электронный источник: Вельбицкий И.В., Графическое программирование только одной схемой (дугой) без операторов типа goto, if, for, ... без меток и скобок типа begin-end, {-} или 5.6, 55.2, 15.4 преимуществ, http://glushkov.org/wp-content/Velbit-140314.pdf
14. Электронный источник: Вельбицкий И.В., Единая графическая оболочка для всех языков и систем, позволяющая программировать всем, не только программистам, http://glushkov.org/wp-content/Single%20graphical.pdf
15. Электронный источник: Вельбицкий И.В., Единая универсальная графическая оболочка для записи программ на любых языках. Визуальная технология программирования нового поколения, http://glushkov.org/wp-content/gr.pr.ed.gr.obol.pdf