ОМСКИЙ ГОСУДАРСИВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

В.В. Кручинин

Практикум по программированию на языке программирования Си

ТОМСК – 2006

Федеральное агентство по образованию

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра промышленной электроники

В.В. Кручинин

Практикум по программированию на языке программирования Си

2006

Кручинин В.В.
Практикум по программированию на языке программирования Си. — Томск: Томский государственный университет систем управления и радиоэлектроники. — 160 с.

ОГЛАВЛЕНИЕ

— Как ты сюда попал?
— Через дырку в заднем проходе!
Из разговора двух хакеров.

1 АЛГОРИТМЫ АРИФМЕТИКИ

1.1 Проблемы целочисленной арифметики

Рассмотрим пример вычисления чисел Фибоначчи, которые вычисляются по следующей формуле

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

Очевидно, что все числа Фибоначчи целые неотрицательные числа. Вот первые несколько чисел

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377 …

Ниже представлена программа для вычисления.

Макроопределение NUMBER задает тип целого в Си это (char — 1 байт, short — 2 байта, int — 4 байта, long — 4 байта, __int64 — 8 байт).

Макроопределение ***FORMAT*** — описывает формат вывода в функции printf (для целых типов char, short, int, long — "%d ", для __int64 — "%Ld ")

Функция Fibo(n) возвращает вычисленное число по номеру ***n***.

Алгоритм следующий:

Вычисления производятся с начального значения i1=1 и i2=1.

Далее вычисляется следующее число Фибоначчи, переменные i1 и i2 корректируются.

Процесс повторяется пока, счетчик не достигнет числа n.

```
#define NUMBER int
#define FORMAT    "%d \n"
NUMBER Fibo(int n) {
 NUMBER i1, i2, f;
  int j;
  if(n<2) return 1;
  i1=1;
  i2=1;
```

```
  for(j=2; j<n; j++)
   {
     f=i1+i2;
     i1=i2;
     i2=f;
     printf(FORMAT, f);
   }
  return f;
}
```

Рассмотрим теперь проблемы вычисления, используя различные типы целых, определенных в Си. Подставляя различные типы в макроопределение NUMBER и соответствующий формат в макроопределение FORMAT. Попытаемся вычислить число Fibo(100), мы обнаружим, что наступает момент, когда два предыдущих значения положительные, а результат отрицательный. Это означает, что происходит переполнение. Этот факт говорит о том, что нужно внимательно выбирать тип представления целого числа.

```
void main(){
 printf("размер целого типа  %d \n",sizeof(NUMBER));
 printf(FORMAT,Fibbo(100));
}
```

Другой простой пример: необходимо подсчитать количество дней между двумя датами, при это логично предложить 16 разрядное представление целого числа. Однако, если это число будет знаковое, то максимальное значение $2^{15}-1$=32767, и если, например, разность будет равна 100 лет то приблизительное значение будет равно 100*365=36500, отсюда при вычислении такой разности произойдет переполнение.

1.2 Проблемы плавающего формата

Для представления вещественных чисел разработан вещественный формат. Этот формат предполагает, что (о математике плавающей запятой) в Си имеется следующие типы плавающей запятой

float, double, long double.

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

```
void main( void )
{
  double a = 0.10000001;
  /* Show original control word and do calculation. */
  printf( "Original: 0x%.4x\n", _control87( 0, 0 ) );
  printf( "%1.8f * %1.8f = %.15e\n", a, a, a * a );

  /* Set precision to 24 bits and recalculate. */
  printf( "24-bit:   0x%.4x\n", _control87( PC_24, MCW_PC ) );
  printf( "%1.8f * %1.8f = %.15e\n", a, a, a * a );

  /* Restore to default and recalculate. */
  printf( "Default:  0x%.4x\n",
       _control87( CW_DEFAULT, 0xfffff ) );
  printf( "%1.8f * %1.8f = %.15e\n", a, a, a * a );
  getch();
}
```

Листинг работы программы (Cbuilder-5, Windows-XP)

```
Original: 0x1372
0.10000001 * 0.10000001 = 1.000000200000010e-02
24-bit:   0x1072
0.10000001 * 0.10000001 = 1.000000121208832e-02
Default:  0x825ff
0.10000001 * 0.10000001 = 1.000000200000009e-02
```

Внимательно изучите листинг умножение чисел одинаковое — результат вычислений разный!

Если у Вас другая ОС или система программирования, то результат может другим. Отсюда результаты моделирования или расчеты могут отличаться друг от друга.

1.3 Пользовательская арифметика

Как видим из описаний предыдущих разделов, стандартные арифметические операции имеют существенные ограничения. В некоторых случаях полезно разработать свое представление числа и соответственно свои арифметические операции. Ниже приведено описание такого подхода.

Целое беззнаковое число представляется в виде нового типа определенного с помощью typedef. Структура BigNumberTag содержит два поля: первое поле описывает количество цифр в поле number. В каждом байте поля number может храниться только одна десятичная цифра. Причем десятичное сило записывается с права на лево, т.е. самый младший разряд находится слева. Ниже записана структура числа и некоторые арифметические операции. Здесь представлены алгоритмы сложения и умножения, основанные на школьной арифметике сложения и умножения столбиком. Кроме того, имеется функция вычисления с пока алгоритма умножения столбиком и функция вычисления факториала.

Особенность такого представления, учет длины числа. Ниже приводятся: описание структуры и функции.

```
#define SIZEBIGNUMBERS 200   //размер числа в цифрах
//описание длинного целого беззнакового числа
typedef struct BigNumberTag {
 int size;  //количество цифр в числе
 char number[SIZEBIGNUMBERS];   //память под десятичное число
} BigNumber;

//установить значение числа используя строку символов
int SetBigNumber(BigNumber *bn, char *num){
 int i,j;
 if((bn->size=strlen(num))>SIZEBIGNUMBERS) return -1; // слишком
большое число
 for(i=bn->size-1, j=0; i>=0; i--,j++) //храним число в обратном порядке
  bn->number[j]=num[i]-'0';  //храним не коды цифр а сами цифры
}

//печать числа
void PrintBigNumber(BigNumber *bn){
 for(int i=bn->size-1; i>=0; i--)
  printf("%c",bn->number[i]+'0');
 printf("\n");
}

//копировать значение числа dst=src
void CopyBigNumber(BigNumber *dst, BigNumber *src){
  dst->size=src->size;
  for(int i=0; i<dst->size; i++)
   dst->number[i]=src->number[i];
```

```
}

//Сложить bn1 и bn2 результат поместить в bn1
int AddBigNumber(BigNumber *bn1,BigNumber *bn2){
  BigNumber res;  //здесь храним резудьтат сложения
  int d;          //результат сложения двух чисел
  int carry=0;    //хранит единицку переноса
  BigNumber *min,*max; //вспомогательные указатели
  if(bn1->size<bn2->size){  //определяем длинное число и короткое
   min=bn1;  //короткое
   max=bn2;  //длинное
   }
  else{
   min=bn2;
   max=bn1;
   }
  //поскольку цифры храним в обратном порядке, начинаем сложение с левой цифры
  for(int i=0; i<max->size; i++){
     //если позиция не превысила короткого слова то производим сложение
     if(i<min->size) d=min->number[i]+max->number[i]+carry;
    else d=max->number[i]+carry;  //в противном случае учитывается только перенос
     if(d>9){   //установливаем перенос в другой разряд
       res.number[i]=d-10;
       carry=1;
       }
     else {     //здесь переноса нет
       res.number[i]=d;
       carry=0;
     }
    } //завершение цикла
  if(carry){  //если после цикла еденица переноса установлена, то увеличиваем размер результирующего числа
    if(max->size<SIZEBIGNUMBERS){ //здесь проверяем выход за границу (переполнение)
     res.number[max->size]=carry; //запомнить перенос
     res.size=max->size+1;        //увеличить размер числа
     }
    else return -1;               //выход по переполнению
    }
```

```
   else res.size=max->size;       //установить размер
   CopyBigNumber(bn1,&res);
  return 1;
}

//функция сдвига вправо (умножение на 10)
//bn - число для сдвига
//count - количество сдвигов
int ShiftRight(BigNumber *bn,int count){
  if(bn->size+count<SIZEBIGNUMBERS){  //если нет переполнения
    for(int i=bn->size-1; i>=0; i--){ //копировать с конца на count разрадов
      bn->number[i+count]=bn->number[i];
      }
    bn->size+=count; //изменить длину числа
    for(int i=0; i<count; i++) bn->number[i]=0; //заполнить нулями слева
    return 1; //нормальное завершение
  }
  return -1;  //переполнение
}

//функция умножения числа на цифру
int MultDigit(BigNumber *res, BigNumber *bn,int digit){
  int d;
  int carry=0;
  if(digit==0) { //если цифра равна 0
   res->size=1;
   res->number[0]=0;
   return 1;
  }
  for(int i=0; i<bn->size; i++){
   d=bn->number[i]*digit;   //умножаем разряд на цифру
   if(d%10+carry<10){       //если текущий разряд не переполняется
      res->number[i]=d%10+carry;  //вычисляем значение текущего разряда
числа
      carry=d/10;               //вычисляем значение переноса в следующий разряд
      }
   else{                    //переполнение текущего разряда
     res->number[i]=d%10+carry-10; //определяем значение текущего разряда
     carry=d/10+1;               //вычисляем общий перенос
     }
  }
```

```
  if(carry){ //если меняется размер слова
    if(bn->size<SIZEBIGNUMBERS){ //переполнение ?
     res->number[bn->size]=carry;  //запоминаем перенос
     res->size=bn->size+1;         //увеличиваем размер числа
     }
    else return -1; //выход с переполнением
    }
   else res->size=bn->size;  //запоминаем размер
   return 1;   //норамальное завершение
}

//функция умноженя больших чисел res=bn1*bn2 ()
int MultBigNumber(BigNumber *res,BigNumber *bn1, BigNumber *bn2){
  BigNumber tmp;
  for(int i=0; i<bn2->size; i++){
   MultDigit(&tmp,bn1,bn2->number[i]); //умножаем текущий разряд числа
bn2 на число bn1
   if(i==0) CopyBigNumber(res,&tmp);   //если первый раз запоминаем ре-
зультат в res
   else{ //сдвигаем результат на i позиций вправо (умножение на 10*i)
    ShiftRight(&tmp,i);
    //PrintBigNumber(&tmp);
    AddBigNumber(res,&tmp);  //и складываем с res
    }
   }
  return 1;
}

//функция умножения больших чисел res=bn1*bn2 и вывод столбиком
int MultBigNumberWithPrint(BigNumber *res,BigNumber *bn1, BigNumber
*bn2){
  int len;
  BigNumber tmp;
  int row=2;
  clrscr();
  gotoxy(len/2,1);
  cprintf("Multiple from Kru\n");
  len=(bn1->size+bn2->size+2);
  for(int i=0; i<len; i++){
    gotoxy(len-i,row+1);
    if(i<bn1->size) cprintf("%c",bn1->number[i]+'0');
```

```
    gotoxy(len-i,row+2);
    if(i<bn2->size) cprintf("%c",bn2->number[i]+'0');
   }

  for(int i=0; i<((bn1->size>bn2->size)?bn1->size:bn2->size); i++){
   gotoxy(len-i,row+3);
   cprintf("-");
   }
  int i;
  for(i=0; i<bn2->size; i++){
   MultDigit(&tmp,bn1,bn2->number[i]); //умножаем текущий разряд числа
bn2 на число bn1
   for(int j=0; j<len; j++){
     gotoxy(len-j-i,row+4+i);
     if(j<tmp.size) cprintf("%c",tmp.number[j]+'0');
    }
   if(i==0) CopyBigNumber(res,&tmp);   //если первый раз запоминаем ре-
зультат в res
   else{ //сдвигаем результат на i позиций вправо (умножение на 10*i)
    ShiftRight(&tmp,i);
    //PrintBigNumber(&tmp);
    AddBigNumber(res,&tmp);  //и складываем с res
    }
   }
   for(int j=0; j<len; j++){
   gotoxy(len-j,row+4+i);
   cprintf("-");
  }
   for(int j=0; j<len; j++){
     gotoxy(len-j,row+5+i);
     if(j<res->size) cprintf("%c",res->number[j]+'0');
    }
   return 1;
}

//вычиление факторила
void Factorial(BigNumber *res,int n){
 BigNumber inc, cur, tmp;
 SetBigNumber(&inc,"1");  //запоминаем единицу
 SetBigNumber(&cur,"1");  //текущее значение аргумента
 SetBigNumber(res,"1");   //текущее значение факториала
```

```
  for(int i=1; i<=n; i++){ //цикл
    MultBigNumber(&tmp,res,&cur); //tmp=cur*res
                          //здесь нельзя res=res*cur
    CopyBigNumber(res,&tmp);      //res=tmp
    PrintBigNumber(res);          //печать промежуточного результат
    AddBigNumber(&cur,&inc);      //cur=cur+1
  }
}

int main(int argc, char* argv[])
{
  BigNumber bn, bn2, res;
  SetBigNumber(&bn,"993145678399");
  SetBigNumber(&bn2,"9991");
  //PrintBigNumber(&bn2);
  //AddBigNumber(&bn,&bn2);
  MultBigNumberWithPrint(&res,&bn,&bn2);
  //PrintBigNumber(&bn);
  //PrintBigNumber(&res);
  //Factorial(&res,10);
  getch();
  return 0;
}
```

Упражнения

1. Написать функцию вычисления разности двух чисел типа BigNumber.

2. Написать функцию деления двух чисел типа BigNumber.

3. Написать функцию вывода алгоритма деления в виде столбика.

4. Разработать алгоритмы знаковой арифметики.

5. Разработать алгоритмы вещественной арифметики (указание: необходимо в структуру BigNumber добавить позицию десятичной запятой (точки) и корректировать ее в арифметических операциях.

Литература по описанию арифметических операций

Гуртовцев А.Л., Гудыменко С.В. Программы для микропроцессоров: Справ. пособие. — Мн.: Высш. шк., 1989. — 352 с.

1.4 Алгоритмические проблемы

Предположим, Вы запрограммировали некоторый алгоритм. Программа на простых примерах работает прекрасно. Однако, подставляя реальные данные, программа начинает считать … 5 мин, 10 мин .. 1 час … Вы задачу снимаете и начинаете анализировать на предмет зацикливания. Но зацикливания нет. Теперь очередь настает анализу алгоритма. Программист пытается оценить время выполнения алгоритма. Рассмотрим конкретный пример, предположим имеется следующий фрагмент программы:

```
unsigned __int64 i, k=0Xffffffffffffffff;
for(i=0; i<k; i++);
printf("Hello student!!!\n"); //Интересно, сколько времени ждать привета?
```

Сколько времени займет вычисление, попробуйте на своем компьютере!

Ниже приведен текст программы, который позволяет сделать приблизительные расчеты.

```
#include <conio.h>
#include <stdio.h>
void main(){ //подсчет времени работы фрагмента программы
unsigned __int64 i, k=0Xffffffffffffffff;
  printf("number of iterations=%Lu\n",k); //число итераций в цикле
 k=k/1000000000000LU; //1 гигагерц (тактовая частота процессора)
 k=k*10; // одна итерация - 10 тактов процессора
  printf("second=%Lu\n",k); //всего секунд
 k=k/3600; //количество часов непрерывной работы
   printf("hour=%Lu\n",k);
 k=k/24; //количество суток
   printf("day=%Lu\n",k);
 k=k/365; //количество лет
   int d=k%365; //итого
 printf("yaers=%Lu days=%d\n",k,d);
 getch();
}
```

Листинг работы программы

```
number of iterations=18446744073709551615
second=184467440
hour=51240
```

day=2135
yaers=5 days=5

Таким образом, это достаточно большое время. Даже увеличивая тактовую частоту в 100 раз, получим 21 день непрерывной работы компьютера.

1.5 Функция Аккермана

Еще один наглядный пример простого алгоритма, но показывающего экспоненциальный взрыв времени вычисления.

Функция Аккермана[2] относится к классу дважды рекурсивных и имеет следующий вид:

$$A(m,n) = \begin{cases} n+1, & m = 0; \\ A(m-1,1), & n = 0; \\ A(m-1, A(m,n-1)), & m > 0, n > 0. \end{cases}$$

Ниже приведена рекурсивная функция подсчета.

```
int Akk(int m,int n){ //реализация подсчета функции Аккермана
if(m==0) return n+1;
if(n==0) return Akk(m-1,1);
return Akk(m-1,Akk(m,n-1));
}
```

Теперь необходимо вычислить Akk(4,2)=? … попробуйте поэкспериментировать!

Запишем явно функцию Аккермана для фиксированных значений m (здесь приведены итоговые формулы, те, кто хочет проверить, может сделать это сам, используя метод математической индукции и формулу геометрической прогрессии)

1) $A(0,n) = n+1$, при m=0;
2) $A(1,n) = n+2$, при m=1;
3) $A(2,n) = 2*n+3$, при m=2;
4) $A(3,n) = 2^{n+3} - 3$, при m=3.

Теперь запишем выражение для A(4,2)

$$A(4,2) = A(3, A(4,1)) = 2^{A(4,1)+3} - 3;$$

$$A(4,1) = A(3, A(4,0)) = 2^{A(4,0)+3} - 3;$$
$$A(4,0) = A(3,1) = 2^{1+3} - 3 = 16 - 3 = 13;$$
$$A(4,1) = 2^{13+3} - 3 = 65536\text{-}3 = 65533;$$
$$A(4,2) = 2^{65533+3} - 3.$$

Вот итоговый факт, для представления результирующего числа необходимо 65 килобит. Соответственно для подсчета такого числа необходимо $2^{65533+3} - 3$ вызовов функции Akk!

Вот это и есть экспоненциальный взрыв!

1.6 Некоторые примеры решения сложных вычислительных задач

Во многих случаях задача может иметь много различных вариантов решения.

Используя общий метод решения данной конкретной задачи можно решить ее, но при этом вычислительная система может быть сильно загружена. Если рассматривать особенности данной конкретной задачи можно получить более эффективные алгоритмы. Ниже описаны два наглядных примера, показывающие такую ситуацию.

Задача № 1

Необходимо подсчитать количество нулей содержащихся в конце числа, вычисленного по формуле факториала. Например, число 3!=6 имеет ноль нулей, число 5!=120 — один ноль, 10!= 3628800 — два нуля.

Понятно, что самым простым решением является написать функцию вычисления факториала, затем полученное число делить его на 10, пока делимое делится нацело, таким образом подсчитать число нулей. Или преобразовать число в строку символов и подсчитать количество символов ‘0’ в конце строки.

При небольших значениях аргумента факториала можно воспользоваться стандартной арифметикой, но при увеличении аргумента приходится перейти на собственную арифметику (см. раздел). Однако для совсем больших чисел, например 10000! или 100000! время вычисления будет слишком велико.

Рассмотрим теперь особенности данной конкретной задачи. Если внимательно изучить появление нулей в конце числа, то можно обнаружить, что ноль появляется, если число в произведении кратно 5. Первый ноль появляется в числе 5!=120, второй ноль в числе 10! Нетрудно подсчитать чис-

ло нулей в числе 100! Их будет 24. Числа дающие в произведении по одному нулю (5, 10, 15, 20, 30, 35, 40, 45, 55, 60, 65, 70, 80, 85, 90, 95), а числа 25, 50, 75, 100 дают по два нуля. Таким образом, алгоритм подсчета нулей в числе n! следующий:

Шаг 1 I=0, s=0.
Шаг 2 I=I+1.
Шаг 3 определяем j как степень кратности числа I числу 5.
Шаг 4 s=s+j.
Шаг 5 переходим на шаг 2, пока не достигнем равенства I=n.

```
//функция подсчета степени кратности 5
int CountZero(int num){
 int i=0;
 while(1){
  if(num%5==0) {
   i++;
   num=num/5;
   }
  else break;
 }
 return i;
}
//подсчет числа нулей в конце числа 10000!
void main(){
       int count=0;
       for(int i=1; i<=10000; i++){
        int z=CountZero(i);
        count=count+z;
        }
       printf("%d\n",count);
 }
```

Задача 1. Найти число, дающее максимальное количество нулей в n!
Задача 2. Найти количество чисел, дающее заданное число нулей.

Задача № 2

Дано число n. Необходимо определить является ли число Фибоначчи F(n) четным.

Решение 1. Вычислить число F(n) и проверить делимость на 2. Однако, учитывая быстрый рост чисел Фибоначчи, такое решение может быть для сравнительно малых чисел.

Решение 2. Если учесть, что четность можно определить по последней десятичной цифре числа, то можно предложить другое решение, которое не требует вычисление всего числа, а только последней десятичной цифры. Алгоритм будет похож на алгоритм вычисления числа Фибоначчи, но только учитывается последняя цифра.

```
//Функция определения четности числа   Фибоначчи(100000);
fibo(){
 int i1=1;
 int i2=1;
 int fi;
 for(int i=0; i<100000; i++){
    fi=i1+i2; //вычисляем суммы двух предыдущих чисел
    if(fi/10) fi=fi%10; //если сумма больше 10, то берем остаток от деления 10
    i1=i2;  //корректируем значения i1 и i2 для вычисления следующего числа
    i2=fi;
   }
  if(fi%2==0) printf("Number fi(100000) is even"); //число четное
else printf("Number fi(100000) is no even"); //число нечетное
 }
```

Задание. Написать функцию, определяющую распределение четных и нечетных чисел Фибоначчи на интервале от [1,100000].

2 ОБРАБОТКА МАТРИЦ

Матрица это таблица чисел. Доступ к элементам матрицы осуществляется по номерам строки и столбца. Часто номер строки и столбца называют индексом. Размерность матрицы определяется по числу строки и столбцов. Тензором называют матрицу, у которой количество индексов может превышать два. Представление данных в форме матриц является стандартным для многих языков программирования. В языке программирования Си имеется возможность обработки матриц и тензоров (примеры см. объявление массивов).

При явном указании размерностей матрицы или тензора память автоматически распределяется транслятором. Однако не всегда возможно заранее определить размерность матрицы. В некоторых случаях необходимо чтобы размерность матрицы можно было бы задавать динамически.

В языке Си такое возможно, однако необходимо знать механизм хранения и распределения памяти. Этот механизм основан на векторизации массивов.

2.1 Динамическое распределение для матрицы целых чисел

Функция распределения памяти и установки начальных значений для целой матрицы

```
int **AllocIntMatrix(int m, int n){
 int **mas;
 int i,j;
mas=(int **)malloc(sizeof(mas)*m);   //выделение памяти под вектор указа-
телей размерность m
 if(mas==NULL) return NULL;
 for(i=0; i<m; i++){                  //выделение памяти под строку целых
чисел
   mas[i]=(int *)malloc(sizeof(int)*n);  //выделяем память под вектор целых
чисел размерность n
   if(mas[i]==NULL) return NULL;
  }
 for(i=0; i<m; i++)                   //начальная инициализация матрицы це-
лых чисел
 for(j=0; j<n; j++)                   //все элементы устанавливаем в едини-
цу
  mas[i][j]=1;
 return mas;
}
```

Функция удаление памяти занимаемой матрицей

```
void DeleteIntMatrix(int **matr, int m){
  int i;
  for(i=0; i<m; i++) free(matr[i]); //освободить память выделенную под стро-
ки
  free(matr); //освободить память выделенную по массив указателей
}
```

Тестовый пример использования целых матриц

```
void main(){
  int i,j,k;
 int **imatr1=AllocIntMatrix(2,3); //распределить память под первую матри-
цу
```

```
 int **imatr2=AllocIntMatrix(3,2); //распределить память под вторую матрицу
 int **res=AllocIntMatrix(2,2);      //распределить память под результирующую
 for(k=0; k<2; k++) //вычислить произведение двух матриц
 for(i=0; i<2; i++) {
   res[k][i]=0;
   for(j=0; j<3; j++)
    res[k][i]+=imatr2[j][i]*imatr1[i][j];
 }
 for(i=0; i<2; i++){ //вывести результат
 for(j=0; j<2; j++)
  printf("%d ",res[i][j]);
  printf("\n");
 }
//освободть память
 DeleteIntMatrix(imatr1,2);
 DeleteIntMatrix(imatr2,3);
 DeleteIntMatrix(res,2);
 getch();
}
```

2.2 Обработка матриц строк символов

Рассмотрим теперь использование матриц для хранения строк символов. Строки символов не фиксированной длины, а переменной. Идея заключается в следующем: память в матрице распределяем только для указателей, т.е. элементом матрицы будет указатель. Тем самым необходимо использовать тройной указатель. Для начала зададим типы.

```
#define TYPE char
#define MATR char*** //тройной указатель
```

Функция выода сообщения об ошибке и завершении выполнения программы. Здесь используется специальная библиотечная функция exit()

```
void ErrorMem(){
 printf("Ошибка: память не выделена\n");
 exit(0);
}
```

Функция создания копии строки в динамической памяти

```
char *CreateStr(char *str){
 char *ptr;
 int len=strlen(str);       //определение количества символов в строке
 ptr=(char*)malloc(len+1);
 if(ptr==NULL) ErrorMem(); //выход если нет памяти
 strcpy(ptr,str);
 return ptr;
}
```

Функция создания матрицы указателей на строки символов

```
TYPE ***GetMem(int m, int n){
 TYPE ***mas;
 int i,j;
 mas=(TYPE ***)malloc(sizeof(mas)*m);    //выделение памяти под вектор
строк
 if(mas==NULL) ErrorMem();
 for(i=0; i<m; i++)  {                        //выделение памяти под указатели на
строки
    mas[i]=(char **)malloc(sizeof(*mas)*n);
   if(mas[i]==NULL) ErrorMem();
  }
 for(i=0; i<m; i++)                    //начальная инициализация матрицы строк
 for(j=0; j<n; j++)
  mas[i][j]=NULL;
 return mas;
}
```

Функция освобождение памяти занимаемой матрицей строк

```
void FreeMem(TYPE ***mas,int m, int n){
 int i,j;
 for(i=0; i<m; i++)   //освобождение памяти занимаемые строками символов
 for(j=0; j<n; j++)  {
   if(mas[i][j]!=NULL) free(mas[i][j]);
  }
 for(i=0; i<m; i++)  //освобождение памяти занимаемые строками матрицы
  free(mas[i]);
 free(mas);          //освобождение памяти занимаемый под вектор строк
}
```

Тестовый пример использования матрицы строк символов

```
void TestStringMatrix(){
 int i, j, k;
 MATR mas;
 char *p1="apple",
      *p2="pear",
      *p3="cucumber",
      *p4="tomato";
 mas=GetMem(2,2);           //динамическое выделение памяти под
матрицу(2,2)
 mas[0][0]=CreateStr(p1);   //присвоение строк элементам матрицы
 mas[0][1]=CreateStr(p2);
 mas[1][0]=CreateStr(p3);
 mas[1][1]=CreateStr(p4);
 for(i=0; i<2; i++)        //использование матрицы
 for(j=0; j<2; j++){
  for(k=0; k<4; k++)
  printf("%c",mas[i][j]);
  printf("\n");
  }
 FreeMem(mas,2,2);         //освобождение памяти занимаемой матрицей
 getch();
}
```

2.3 Механизм матрицы как контейнера

```
#define MatrixContainer void ***
void ***AllocMatrixContainer(int m, int n){
 void ***mas;
 int i,j;
 mas=(void ***)malloc(sizeof(mas)*m);    //выделение памяти под вектор
строк
 if(mas==NULL) ErrorMem();
 for(i=0; i<m; i++)                      //выделение памяти вектора
указателей
  {
   mas[i]=(void **)malloc(sizeof(*mas)*n);
   if(mas[i]==NULL) ErrorMem();
  }
```

```
 for(i=0; i<m; i++)                    //начальная инициализация матрицы
указателей
 for(j=0; j<n; j++)
  mas[i][j]=NULL;
 return mas;
}

void DeleteMatrix(MatrixContainer matrix,int m, int n,void(*DeleteElem)(void*
elem)){
 int i,j;
 for(i=0; i<2; i++)
 for(j=0; j<3; j++)
  DeleteElem(matrix[i][j]); //удалить все элементы
 for(i=0; i<2; i++) free(matrix[i]); //удалить строки
 free(matrix);  //удалить массив указателей
}
```

Пример использования. Необходимо хранить матрицу точек, заданных координатами.

Описание точки дано ниже.

```
typedef struct Point_ {
 int x;
 int y;
}  MPoint;
```

Функция создания структуры MPoint

```
MPoint *CreatePoint(int x, int y){
 MPoint *point=(MPoint *)malloc(sizeof(MPoint));
 point->x=x;
 point->y=y;
}
```

Ниже представлен пример функции использования матрицы-контейнера хранящего множество точек.

```
void TestContainer(){
 MatrixContainer m=AllocMatrixContainer(2,3); //распределение паями под
контейнер
 int i,j;
 for(i=0; i<2; i++) //присвоение значений указателем контейнера на точки
```

```
for(j=0; j<3; j++)
 m[i][j]=CreatePoint(i,j);

for(i=0; i<2; i++) //вывод матрицы точек
for(j=0; j<3; j++)
printf("%d %d\n",((MPoint*)m[i][j])->x,((MPoint*)m[i][j])->y);
DeleteMatrix(m,2,3,free); //удаление матрицы точек
getch();
}
```

3 ОБРАБОТКА СТРОКА СИМВОЛОВ

Владение техникой обработки текстовой информации является неотъемлемым атрибутом современного арсенала программиста. При создании программ текстовой обработки необходимо определить:

1) способы кодирования теста;
2) способы представления строк символов;
3) алгоритмы обработки.

3.1 Способы представления строк символов

Известны следующие способы представление строк символов:

1) в ячейках последовательно представлены коды символов, в последней ячейке содержится некоторый код, который означает конец строки;
2) размер строки задается в начальной ячейке, далее в ячейках последовательно представлены коды символов.

Первоначально ячейкой для представления символов являлся байт. В Си была принята первая форма представления строк символов, концевым значением было принято нулевое значение (‘\0’). Второй способ представления характерен для Паскаля. Оба этих представления имеют недостатки: в первом случае размер строки приходится вычислять, во втором случае строка не может превышать размер 256 символов. Во втором случае приходится вводить понятие коротких и длинных строк, поскольку ячейка, указывающая размер строки, может иметь разный размер (1 байт, слово).

3.2 Алгоритмы обработки

Рассмотрим несколько функций для обработки строк, представленных в виде последовательности байт с нулем в последнем байте.

1. Функция подсчета символов в строке, не считая байта с нулевым значением.

```
int strlen(char *str){
 int i=0;
 while(*str++) i++;
 //for(i=0; *str; str++) i++; //второй вариант реализации
 return i;
}
```

2. Функция подсчета количества слов в предложении, записанного как строка символов.

Разделителем между словами являются пробелы ... количество пробелов неограниченно.

```
int SizeWord(char *express)
{
 int size=0;
 while(*express==' ') express++;  //пропустить пробелы в начале строки
 while(1)  {     //бесконечный цикл
  if(*express==' ')   {  //это пробел
     size++;  //увеличить  счетчик слов
     while(*express==' ') express++; //пропустить пробелы
     if(*express) break;  //выход если конец строки
     }
  else
   if(*express) {  //если это конец строки, конец строки следует сразу за словом
      size++;  // увеличить  счетчик слов
      break;  //выход из цикла
      }
  else
   express++;   //это не пробел и не конец строки, увеличить указатель на следующий символ
 }
 return size;  //вернуть число слов
}
```

Реальной ситуации для разбора теста нужно учитывать значительно больше условий, это всевозможные символы: перевод строки, возврат каретки, табуляция, всевозможные разделители и прочее.

3. Функция сравнения строк на равенство.

```
int strcmp1(char *str1, char *str2) {
 while(str1*)  {
  if(*str1++!=*str2++) return 0;
 }
 return 1;
}
```

4. Функция лексикографического сравнения двух строк.

Эта функция является стандартной для многих системных библиотек. Эта функция возвращает: 1, если str1>str2; -1, если str1<str2, 0, если str1 равна str2.

```
int strcmp2(char *str1, char *str2) {
 while(1) { //бесконечный цикл
  if(*str1>*str2) return 1; //если код текущего символа первой строки больше чем код текущего
                            //символа второй строки, то выход
 else //иначе
 if(*str1<*str2) return -1; //если код текущего символа первой строки меньше чем код текущего
                            //символа второй строки, то выход

 else  //иначе
   if(!*str1) return 0;  //если они равны и это конец строки выход
  str1++;  //увеличить указатели, продолжить цикл.
  str2++;
 } //конец цикла
}
```

5. Функция вставки символа в строку.

Приведенная ниже функция Insert производит вставку символа в первую позицию в строке.

Входными параметрами являются: указатель на начало строки и вставляемый код символа.

```
void Insert(char *str, char sym){
 int len=strlen(str); //определить длину строки
 for(int i=len; i>0; i--) str[i]=str[i-1];  //копировать все символы строки на
                                            //одну позицию вправо начиная с конца строки
 *str=sym;                                  //в нулевую позицию занести символ
}
```

Примеры использования функции:

Пример 1.

```
char str[10]=”xyz”;
Insert(str, ’o’);
printf(“%s”,str); //будет выведено oxyz
```

Пример 2.

```
char helo[10]="helo";
 Insert(&helo[2],'l'); //вставить символ во вторую позицию строки
 printf("%s",helo);  //будет выведено hello
```

6. Функция удаления символа в строке.

```
void DeleteFirstSymbol(char *str){
 int i;
 int len=strlen(str);
 for(i=0; i<len; i++) str[i]=str[i+1];
}
```

7. Функция замены в строке символа табуляции на заданное число пробелов.

str — указатель на строку символов;

size — количество пробелов.

```
int ReplaceTab(char *str,int size){
 int i,j;
 int len=strlen(str); //вычислить длину строки
 for(i=0; i<len; i++) { //цикл просмотра символов в строке
   if(str[i]=='\t')  { //найден символ табуляции
      str[i]='  ';  //заменить символ табуляции на пробел
      for(j=0; j<size-1; j++) Insert(&str[i],'  '); //вставить недостающие пробелы
      len+=size-1; //корректировать длину строки
      i+=size-1;     //корректировать текущую позицию
    }
  }
  return len;  //вернуть реальную длину строки
}
```

8. Функция проверки вхождение подстроки в строку.

```
char *ch; //глобальный указатель
void InitCh(char *str){ //функция инициализации
 ch=str;
}
void SkipBlanks() {  //функция пропуска пробелов
   while(*ch==' ') ch++;
}
```

```
int match(char *tst) { //функция нахождения вхождения подстроки tst в строке ch
 char *chp;
 SkipBlanks();
 chp=ch; //инициализация
  while(*tst) { //цикл просмотра по длине tst
  if(*tst++!=*chp++) return 0;  //не входит
 }
 ch=chp; //входит, указатель перемещаем на символ, следующий за найденной подстрокой
 return 1;
}
```

Пример разбора предложения

```
void SimpleParse(char *Express){ //функция разбора предложения
 InitCh(Express);
 printf("<разбор предложения %s>\n",Express);
 if(match("Вася")||match("Петя")||match("Маша")) printf("<подлежащее>");
 printf("+");
 if(match("взял")||match("получил")||match("написала")) printf("<сказуемое>");
 printf("+");
 if(match("письмо")||match("сочинение")||match("записку")) printf("<дополнение>");
}

SimpleParse("Маша написала письмо");
```

<разбор предложения Маша написала письмо >
<подлежащее>+<сказуемое>+<дополнение>

9. Функция преобразования строки символов в число.

Пусть дано десятичное число «356», представленное в виде строки, тогда алгоритм вычисления может быть следующий:

(10*(10*(10*0+’3’-‘0’)+’5’-‘0’)+’6’-‘0’)

Здесь применяется свойство кодирования цифр: цифры кодируются последовательно, по возрастанию. Например см. ASCII таблицу. Поэтому ‘5’-‘0’=5.

```
int CharToInt(char *num){
 int rez=0;
 int sign=1; //по умолчанию число положительное
 if( *num=='-' ) {
   sign=-1;  //число отрицательное
   num++;  //пропустить знак
   }
else
 if(*num=='+')  num++;  //пропустить знак
 while(1) { //цикл просмотра символов цифр
  if(*num<'0' || *num>'9') return rez; //это не цифра, вернуть число
  rez=10*rez+(*num-'0'); //преобразовать код цифры в число, произвести вычисление
  num++; //просмотреть следующий символ
 }
}
```

Задание на контрольную работу

1. Разобраться что делает каждая программа и сделать описание.
2. Написать функцию, которая находит в предложении слово максимальной длины.
3. Написть функцию, которая сортирует слова в преложении в лексикографическом порядке.
4. Написать преобразования целого числа в строку символов.

4 ОБРАБОТКА ЛИНЕЙНЫХ СПИСКОВ

Списки являются мощным инструментом представления информации, структура которой может динамически меняться. Список обычно состоит из заголовка и множества связанных элементов списка. Заголовок хранит информацию о списке в целом: количество элементов, адреса специально выделенных элементов, имя владельца, время создания и т.д.

Элемент списка это блок данных, который разбивается на две части: адресную, предназначенную для навигации по списку и часть описывающую данные. Рассмотрим организацию простейшего списка (см. рис.)

4.1 Линейный одно-связанный список

Описание элемента списка

```
struct Link {
   struct Link *next;
   int data;
};
```

Описание заголовка списка

```
struct SimpleList {
  struct Link *first;
  int len;
};
```

Функция инициализации списка

```
void InitSimpleList(struct SimpleList *list){
 list->first=NULL;
 list->len=0;
}
```

Функция вставки нового элемента данных в список

```
void AddSimpleList(struct SimpleList *list, int data){
 struct Link *link=(struct Link*)malloc(sizeof(struct Link));
 link->data=data;
 link->next=list->first;
 list->first=link;
```

```
 list->len++;
}
```

Функция печать списка

```
void PrintSimpleList(struct SimpleList *list){
 struct Link *link;
 printf("++PrintList++\n");
 for(link=list->first; link!=NULL; link=link->next)
  printf(" %d",link->data);
 printf("\n");
}

//Пример использования списка
void main() {
  struct SimpleList Numbers;
  InitSimpleList(&Numbers);
  AddSimpleList(&Numbers,100);
  AddSimpleList(&Numbers,70);
  AddSimpleList(&Numbers,500);
  AddSimpleList(&Numbers,10000);
  AddSimpleList(&Numbers,2200);
  PrintSimpleList(&Numbers);
  getch();
}
```

4.2 Линейный двухсвязный список для хранения строк символов

1. Описание типа элемента списка

```
typedef struct ListElTag {
   struct ListElTag *left;   //связь влево
   struct ListElTag *right; //связь вправо
   char *name;                  //хранимый указатель на строку символов
   } LISTEL;
```

2. Описание типа списка

```
typedef struct ListTag {
  LISTEL *head;         //указатель на голову
  LISTEL *tail;          //указатель на хвост
```

```
 int size;                  //количество элементов
 } LIST;
```

3. Функция инициализации списка

```
void InitList(LIST *list){
 printf("инициализация списка\n");
 list->head=list->tail=NULL;
 list->size=0;
}
```

4. Функция добавления новой строки в список

```
void AddList(LIST *list,char *name){
 int len=strlen(name);
 //выделение памяти
 LISTEL *listel=(LISTEL*)malloc(sizeof(listel)); //выделение памяти под
новый элемент
 if(listel==NULL) return;
 listel->name=(char*)malloc(len+1);          //выделение памяти по строку
 if(listel->name==NULL) return;
 strcpy(listel->name,name);                   //копировать строку
 listel->left=NULL;
 listel->right=NULL;
 printf("добавление элемента\n");
 //организация связи
 if(list->head==NULL) {//список пуст
    list->head=list->tail=listel;
  }
 else { //добавление элемента в конец списка
   list->tail->left=listel;
   listel->right=list->tail;
   list->tail=listel;
  }
  list->size++;
}
```

5. Удаление строки из списка

```
void DeleteEl(LIST *list){
```

```
 LISTEL *ptr;
 printf("удаление элемента\n");
 if(list->head==NULL) return; //список пуст
 ptr=list->head;
 if(ptr==list->tail)  { //в списке один элемент
    list->head=list->tail=NULL;
   }
 else {  //в списке несколько элементов
   list->head=ptr->left;
   list->head->right=NULL;
  }
  list->size--;
  free(ptr);
}
```

6. Печать всех строк списка

```
void OutList(LIST *list){
 LISTEL *ptr;
 printf("вывод списка\n");
 for(ptr=list->head; ptr!=NULL; ptr=ptr->left)
  printf("%s\n",ptr->name);
}
```

7. Функция удаления всех элементов списка

```
void DeleteAll(LIST *list) {
 LISTEL *ptr, *tmp;
 printf("удаление списка\n");
 for(ptr=list->head; ptr!=NULL;)
  {
   tmp=ptr->left;
   free(ptr->name);
   free(ptr);
   ptr=tmp;
  }
}
```

8. Механизм перечисления элементов списка

Просмотр списка можно сделать универсальным. Для этого в функцию перечисления списка можно передать указатель на функцию обработки

данных. Как правило, такие функции имеют название ForEach и одним из параметров имеет указатель на функцию — обработчик.

В качестве простейшего примера можно рассмотреть организацию печати элементов списка.

Для этого записывается функция вывода строки (Out).

```
void Out(char *str) {
 printf("++++ %s ++++\n",str);
}

void ForEach(LISTEL *beg, LISTEL *end, void (*func)(char *)){
 LISTEL *ptr;
  for(ptr=beg; ptr!=end; ptr=ptr->left)   func(ptr->name);
 }
```

Тогда передавая два указателя beg и end и функцию Out можно отпечатать фрагмент списка

Например,

```
List Names;
… //инициализация и добавление
ForEach(Names->head,NULL,Out); //печать содержимого списка
```

В некоторых случаях полезна функция, которая принимает указатель на функцию с двумя параметрами, первый параметр — содержит указатель на данные, которые будут переданы из элемента списка. Другой параметр необходим для накопления некоторой информации. Например, необходимо подсчитать число строк, размер которых превышает size. Ниже записан пример.

```
typedef struct { //описание типа второго параметра
  int size;   //задает длину сравнения
  int count;  //число строк превышающих size
} Param;
```

9. Функция подсчета

```
void  CoutString(char *str, void *param){
Param *p=(Param *)param; //преобразование void* в тип Param
if (strlen(str)>=p->size)  p->count++;
}
```

10. Функция перечисления

```
void ForEachParam(LISTEL *beg, LISTEL *end, void (*func)(char *,void
*param ), void *param)
{
 LISTEL *ptr;
  for(ptr=beg; ptr!=end; ptr=ptr->left)   func(ptr->name,param);
 }
```

11. Пример использования

```
List Enimals;
Param NameGre5; //структура для подсчета
NameGre5->size=5; //установка значения границы
NameGre5->count=0; //обнуление счетчика
… //инициализация и добавление в список
ForEach(Enimals ->head,NULL,CountString, (void*)&NameGre5); //подсчет
количества строк >= 5.
printf(“%d”, NameGre5->count);
….
```

12. Поиск в списке

Поиск в списке можно сделать на основе механизма ForEach. Однако этот механизм неэффективен из-за того, что нельзя прервать просмотр, если найден элемент. Ниже предлагается функция нахождения заданной строки. Если строка найдена, то возвращается адрес элемента списка, содержащую эту строку. В противном случае возвращается значение NULL.

```
LISTEL* FindName(LIST *list, char *name) {
 LISTEL *ptr;
 printf("поиск имени\n");
 for(ptr=list->head; ptr!=NULL; ptr=ptr->left)
  if (strcmp(name,ptr->name)==0)  return ptr;
 return NULL;
}
```

13. Удаление элемента списка содержащего искомую строку.

```
int DeleteName(LIST *list, char *name)
{
```

```
 LISTEL *ptr;
 printf("удаление имени %s\n",name);
 if((ptr=FindName(list,name))==NULL) //указанное имя не найдено
    return 0;
 if(ptr==list->head&& ptr==list->tail) { //удаление единственного элемента
     list->head=list->tail=NULL;
  }
 else
 if(ptr==list->tail) { //удаление  хвоста
   list->tail=list->tail->right;
   list->tail->left=NULL;
  }
 else
  if(ptr==list->head) {//удаление головы
    list->head=list->head->left;
    list->head->right=NULL;
   }
 else  { //удаление из середины
    ptr->left->right=ptr->right;
   ptr->right->left=ptr->left;
 }
 free(ptr);
}
```

14. Функция сортировки элементов списка в лексикографическом порядке

```
void SortName(LIST *list) {
 int key=1; //ключ сортировки
 char *tmp;
 LISTEL *ptr;
 printf("сортировка\n");
 while(key) { //сортируем пока есть хотя бы одна перестановка
  key=0; //сбрасывает ключ сортировки
  for(ptr=list->head; ptr!=list->tail; ptr=ptr->left)
   {
    if(strcmp(ptr->name,ptr->left->name)<0) { //сравниваем строки двух соседних элементов
      tmp=ptr->name;      //переставляем местами указатели на строки
      ptr->name=ptr->left->name;
      ptr->left->name=tmp;
```

```
      key=1; //перестановка есть, сортировка не окончена!
     }
   }//конец for
  } //конец while
} //конец SortName
```

15. Функция построения списка слов из строки символов

```
void MakeListWord(char *exp,LIST *list) {
  char *begword, *word;
  int len;
  int i;
  while(1)  {
   while(*exp==' ') exp++; //пропустить пробелы
   begword=exp; //устанавливаем указатель на начало слова
   len=0;
   while(*exp)  {  //определение длины слова
    if(*exp!=' ')     { //пока не пробел смещаем указатели
      len++;
      exp++;
     }
    else break; //выход если конец строки или пробел
   }//конец while
  if(len==0) return;  //завершение формирования списка
  //выделение слова
  word=(char*)malloc(len+1);
  for(i=0; i<len; i++) word[i]=begword[i];
  word[len]='\0';
  //запись слова в список
  AddList(list,word);
  free(word);
 }
}
```

16. Пример использования функций для работы со списком.

```
void main() {
 LIST list;
 InitList(&list); //проинициализировать список
 AddList(&list,"Mike"); //Добавить несколько строк
 AddList(&list,"Peter");
 AddList(&list,"John");
```

```
 AddList(&list,"Kate");
 OutList(&list); //вывести список
 SortName(&list); //отсортировать список
 OutList(&list); //вывести список
 DeleteName(&list,"Mike"); //удалить несколько строк
 DeleteName(&list,"Peter");
 OutList(&list); //вывести список
 ForEach(list.head,list.head->left,Out); //вывод списка используя механизм
ForEach
 DeleteAll(&list); //удалить список строк
 InitList(&list);// //проинициализировать список
 MakeListWord("Привет Всем Вам !!!",&list); //создать список слов из строки
 OutList(&list); //вывести список
 DeleteAll(&list); //удалить список строк
 getch(); //ожидать ввода символа
}
```

4.3 Понятие контейнера

Под контейнером понимается такая организация хранения данных, при которой возможно хранить разнотипные данные. В контейнер можно складывать строки символов, массивы, различные структуры и прочее. Простейший пример контейнера можно сделать из линейного одно-связанного списка, заменив в описании элемента списка описание данных:

```
typedef struct LinkTag {
  struct LinkTag *next;
  void *data; //это указатель на любые типы данных
};
```

Поскольку функции для работы с контейнером ничего не знают о структуре данных, то функции создания элемента данных, удаления, сравнения, печати и другие пишутся прикладными программистам и затем передаются в качестве параметров в функции типа ForEach.

Основные функции для контейнера следующие:

1) создать контейнер;
2) вставить новый элемент данных;
3) удалить элемент списка;
4) механизм просмотра ForEach(beg,end,Func) и ForEachParam(beg, end, FuncParam, Param);

5) механизм поиска Find(beg, end,Comp) и Find(beg, end, CompParam, Param);
6) механизм сортировки Sort(beg,end, Comp2).
7) удалить контейнер Delete(DeleteElement).

Здесь beg и end указатели на начало и конец поиска. Comp функции сравнения для поиска и сортировки. DeleteElement — функция удаления элемента данных.

Контейнер организованный на основе двухсвязного списка можно найти в файлах KruList.h и KruList.cpp

Переход на идеи объектно-ориентированного программирования дает новые возможности использования контейнеров и будут описаны далее.

4.4 Специальные списки: стек, очередь, дек

Стек — это линейный список, у которого операции вставки и удаления производится с одного конца. Обычно стек имеет один указатель называемый вершиной стека и две функции: Push (вставка на вершину стека) и Pull (удаление элемента на вершине стека). Иногда вводят функцию Top — посмотреть данные на вершине стека.

Очередь — это линейный список, у которого операции вставки и удаления производится с разных концов. Для очереди необходимо хранить два указателя: head — указатель на голову, tail — указатель на хвост. Как известно, очередь растет с хвоста, а уменьшается с головы. Поэтому операция Insert — вставка в очередь производится с хвоста. А операция Delete (Service — обслужить) — с головы.

Дек — это линейный список, у которого операции вставки и удаления производится с обоих концов. Для этого пишется две функции вставки (InsertHead, InsertTail) и две функции удаления DeleteTail и DeleteHead.

Описанный выше односвязный список реализован как стек. А двухсвязный список — это очередь.

Для того чтобы построить свой стек, очередь или дек можно воспользоваться контейнером.

4.5 Достоинства и недостатки списков

Списки имеют свои достоинства и недостатки. Перечислим достоинства:

1. Неизвестно заранее количество элементов в списке.
2. Эффективные операции вставки и удаления

Основные недостатки:

1. Последовательное представление, что приводит к неэффективному поиску.
2. Динамический характер списка приводит также к потере эффективности вычислений.

Недостаток контейнеров: универсальность, дополнительные операции преобразования void указателя, отдельная память для хранения данных. Вызов функций через указатели.

Достоинство контейнеров: универсальность, быстрое создание первой рабочей версии программы.

5 ВВОД И РЕДАКТИРОВАНИЯ ТЕКСТОВЫХ ОБЪЕКТОВ

Введение

Ввод и редактирование текста является одной из важнейших свойств компьютера. было время когда в качестве носителей использовались перфоленты и перфокарты. Для ввода информации существовали отделы подготовки данных, которые по заявкам пользователей вычислительного центра набивали эти перфоленты и перфокарты. Аналогично происходило и редактирование данных.

С появлением ПЭВМ ввод и редактирование данных стало возможным самим пользователем. Используя специальные программы, называемые текстовыми редакторами, пользователь вводил, редактировал и печатал текст.

Рассмотрим варианты разработки простейших программ ввода и редактирования текста.

5.1 Редактор строки

Рассмотрим программу ввода и редактирования строки символов. На входе этой функции передаются: экранные координаты строки и столбца, указатель на строку ввода и максимальный размер строки. Функция организована следующим образом: организуется цикл ввода символа. Если это управляющий символ, то производится преобразование в соответствии с таблицей 1. Преобразование организовано следующим образом: библиотечная функция getch() ожидает ввода очередного символа, если символ управляющий, то эта функция первоначально возвращает ноль, а затем код нажатой клавиши. Отсюда легко предложить следующий алгоритм нумерации управляющих клавиш:

if(!(ch=getch())) ch=getch()+CONTROL;

Читать код символа если это ноль, то чисть следующий код символа и прибвать некоторое смещение CONTROL. Ниже записана таблица определений для управляющих клавиш и их комбинаций.

Таблица управляющих клавиш

```
#define   CONTROL          255          //смещение для разграни-
                                        чения //управляющих
                                        клавиш
```

```
#define   KEY_LEFT        75+(CONTROL)   //стрелка влево
#define   KEY_RIGHT       77+(CONTROL)   //стрелка вправо
#define   KEY_HOME        71+(CONTROL)   //код клавиши Home
#define   KEY_END         79+(CONTROL)   //код клавиши End
#define   KEY_UP          72+(CONTROL)   //код клавиши Стрелка
                                         вверх
#define   KEY_DOWN        80+(CONTROL)   //код клавиши Стрелка
                                         вниз
#define   KEY_BACKSPACE   8              //код клавиши Забой
#define   KEY_DEL         83+(CONTROL)   //код клавиши Delete
#define   KEY_ENTER       13             //код клавиши Ввод
#define   KEY_F1          59+(CONTROL)   //код клавиши F1
#define   KEY_INSERT      82+(CONTROL)   //код клавиши Insert
#define   KEY_ESC         27             //код клавиши Esc
#define   KEY_CTRL_RIGHT  116+(CONTROL)  //код нажатия двух кла-
                                         виш Ctrl и ->
#define   KEY_CTRL_LEFT   115+(CONTROL)  //код нажатия двух кла-
                                         виш Ctrl и <-
#define   KEY_CTRL_Y      25              //код нажатия двух кла-
                                         виш Ctrl и y
#define   KEY_TAB         9              //код клавиши Tab
```

Теперь мы имеем алгоритм распознавания кодов управляющих клавиш и кодов символов.

Ниже в таблице перечислены основные действия, которые выполняются на нажатие управляющих клавиш.

KEY_LEFT	передвинуть курсор влево
KEY_RIGHT	передвинуть курсор вправо
KEY_HOME	передвинуть курсор в начало
KEY_END	передвинуть курсор в конец
KEY_UP	вернуть
KEY_DOWN	вернуть
KEY_BACKSPACE	удалить предыдущий символ
KEY_DEL	удалить текущий символ
KEY_ENTER	Ввод строки
KEY_ESC	отмена ввода, возврат
KEY_TAB	вернуть

Ниже представлен исходный текст функции редактирования строки

```
int KruEditString(int x, int y, char *str, int maxlen){
 int len=strlen(str); //вычислить размер строки
 int pos=0;             //позиция курсора в начале
 int ch;
 gotoxy(x,y);          //переместить курсор экрана в заданную позицию
 cprintf("%s",str);  //вывести строку в позиции курсора
 do{                    //цикл ввода и редактирования
  gotoxy(x+pos,y); //установить курсор в заданную позицию
  if(!(ch=getch())) ch=getch()+CONTROL; //взять введенный код
  switch(ch) //выполнить действие в зависимости от кода
  {
  case KEY_ESC:        return 0; //нажата ESC, вернуть 0
  case KEY_ENTER:   return 1; //нажата клавиша Enter
  case KEY_TAB:        return 2; //нажата клавиша Tab
  case KEY_DOWN:    return 3;//нажата клавиша Down
  case KEY_UP:          return 4;// нажата клавиша Up
  case KEY_HOME:  pos=0; break; //курсор переместить в начала строки
  case KEY_END:   pos=len; break;.// курсор переместить в конец строки
  case KEY_LEFT:  if(pos>0) pos--; break; .// передвинуть курсор влево
  case KEY_RIGHT: if(pos<len) pos++; break;//передвинуть курсор вправо
  case KEY_DEL: //удалить символ
           if(pos<len){
             memmove(&str[pos],&str[pos+1],len-pos+1);
             len--;
             cprintf("%s",&str[pos]);
             putch(' ');
            }
          break;
  case KEY_BACKSPACE: .//удалить предыдущий символ
           if(pos>0) {
             pos--;
             memmove(&str[pos],&str[pos+1],len-pos+1);
             len--;
             gotoxy(x+pos,y);
             cprintf("%s",&str[pos]);
             putch(' ');
            }
    break;
  default: //ввод символа в строку
             if(len<maxlen-1){
             memmove(&str[pos+1],&str[pos],len-pos+1);
```

```
            str[pos]=ch;
            cprintf("%s",&str[pos++]);
            len++;
           }
         break;
  }
 } while(1);
 return 0;
}
```

Обратите внимание на библиотечную функцию memmove, описанную в mem.h. Эта функция копирует правильно даже в тех случаях, когда блоки источника и приемника перекрываются.

При вставке эта функция раздвигает строку в заданной позиции, при удалении сдвигает.

Ниже описан пример вызова редактора строки.

```
void main()
{
  char string[40]="Hello World !!!";
  KruEditString(5,5,string,40);
  gotoxy(5,10);
  printf("Result: %s",string);
  while(!kbhit());
}
```

5.2 Редактор матрицы

Рассмотрим теперь использование идей редактора строки в более сложном примере — ввод и редактирование матрицы. Для простоты размер матрицы зафиксируем и пусть она будет квадратной. Определение SIZE задает размер матрицы.

```
#define SIZE 5
Вводим новый тип CMatrix
typedef double CMatrix[SIZE][SIZE];
```

Функция ввода матрицы KruEditMatrix, на входе координаты левого верхнего угла матрицы, в координатах текстового режима работы терминала. Matrix — матрица размером SIZE.

```
int KruEditMatrix(int x0, int y0,CMatrix Matrix){
 int x=1, y=1;
 int nx=8;
 int i=0, j=0;
 int bkcolor;
 int pos=0, maxlen=8,len=0;
 char str[20]="";
 int ch;
 CPrintMatrix(x0,y0,Matrix);
 i=j=0;
 sprintf(str,"%g",Matrix[i][j]);
 len=strlen(str);
 do{
  gotoxy(x0+x,y0+y);
  textbackground(BLUE);
  cprintf("%-8s",str);
  gotoxy(x0+x+pos,y0+y);
  if(!(ch=getch())) ch=getch()+CONTROL;
  switch(ch)
  {
  case KEY_ESC:
       textbackground(BLACK);
       gotoxy(x0+x,y+y0);
       cprintf("%-8s",str);
       return 0;
  case KEY_ENTER:
       Matrix[i][j]=atof(str);
       textbackground(BLACK);
       gotoxy(x0+x,y0+y);
       cprintf("%-8s",str);
       return 1;
  case KEY_HOME:  pos=0; break;
  case KEY_END:   pos=len; break;
  case KEY_RIGHT: if(pos<len) pos++; break;
  case KEY_CTRL_RIGHT:
      if(j<SIZE-1){
        Matrix[i][j]=atof(str);
        gotoxy(x0+x,y0+y);
        textbackground(BLACK);
        cprintf("%-8s",str);
        j++;
```

```
      x=j*10+1;
      sprintf(str,"%g",Matrix[i][j]);
      pos=0;
      len=strlen(str);
     }
    break;
case KEY_LEFT: if(pos>0) pos--; break;
case KEY_CTRL_LEFT:
    if(j>0){
      Matrix[i][j]=atof(str);
      gotoxy(x0+x,y0+y);
      j--;
      x=j*10+1;
      textbackground(BLACK);
      cprintf("%-8s",str);
      sprintf(str,"%g",Matrix[i][j]);
      pos=0;
      len=strlen(str);
    }
    break;
case KEY_DOWN:
     if(i<SIZE-1){
      Matrix[i][j]=atof(str);
      gotoxy(x0+x,y0+y);
      i++;
      y=2*i+1;
      textbackground(BLACK);
      cprintf("%-8s",str);
      sprintf(str,"%g",Matrix[i][j]);
      pos=0;
      len=strlen(str);
    }
    break;
case KEY_UP:
   if(i>0){
      Matrix[i][j]=atof(str);
      gotoxy(x0+x,y0+y);
      i--;
      y=2*i+1;
      textbackground(BLACK);
      cprintf("%-8s",str);
```

```
      sprintf(str,"%g",Matrix[i][j]);
      pos=0;
      len=strlen(str);
     }
    break;
 case KEY_DEL:
         if(pos<len){
           memmove(&str[pos],&str[pos+1],len-pos+1);
           len--;
           cprintf("%s",&str[pos]);
           putch(' ');
          }
        break;
 case KEY_BACKSPACE:
         if(pos>0) {
           pos--;
           memmove(&str[pos],&str[pos+1],len-pos+1);
           len--;
           gotoxy(x0+x+pos+1,y0+y);
           cprintf("%s",&str[pos]);
           putch(' ');
          }
   break;
 default:
          if(len<maxlen-1&&(isdigit(ch)||ch=='.')){
           memmove(&str[pos+1],&str[pos],len-pos+1);
           str[pos]=ch;
           cprintf("%s",&str[pos++]);
           len++;
          }
        break;
 }
} while(1);
return 0;
}
void CPrintMatrix(int x0, int y0,CMatrix Matrix){
 for(int i=0; i<SIZE; i++)
 for(int j=0; j<SIZE; j++){
  gotoxy(x0+j*10+1,y0+i*2+1);
  cprintf("%g",Matrix[i][j]);
```

```
  }
}

void main()
{
 CMatrix matrix;
 for(int i=0; i<SIZE; i++) //задаем начальные значения
 for(int j=0; j<SIZE; j++)
  matrix[i][j]=0;
 KruEditMatrix(5,2,matrix);
 gotoxy(5,2+SIZE*2);
 cprintf("Result Matrix:");
 CPrintMatrix(5,2+SIZE*2+1,matrix);
 while(!kbhit());
}
```

6 ФАЙЛЫ, БИБЛИОТЕКИ, БАЗЫ ДАННЫХ

6.1 Общие сведения о файловой системе

Обработка файлов одна из фундаментальных основ программирования. Владение техникой программирования обработки файлов является одной из важнейших. Ни одна сколь, ни будь значимая программа не обходится без работы с файлами.

Понятие файла основано на использовании внешних запоминающих устройств, первыми из которых стали магнитные барабаны и ленты. Термин «файл» (в переводе на русский — папка) означает поименованная область внешней памяти, хранящая какие либо данные. Поскольку имена файлов и их атрибуты необходимо где-то хранить появилось понятие каталог. Каталог — это файл, который хранит имена файлов и другие каталоги. Совокупность программ для работы с файлами и каталогами и физическое размещение каталогов и файлов называется файловой системой.

Файловая система является частью операционной системы, и, как правило, являются иерархическими. Это означает, что структуру каталогов можно представить в виде некоторого дерева. Пример организации структуры каталогов показан на рис. В целом файловая система может быть многотомной, т.е. имеется множество разнообразных устройств ввода /вывода на которых могут храниться файлы.

Для указания местоположения файла в файловой системе используется понятие пути (path — тропинка). В путь включаются:

–логическое имя устройства (drive);

–список каталогов;

–имя файла;

–расширение.

Логическое имя устройства в общем случае это любой идентификатор обозначающий некоторое устройство. В DOS или Windows для обозначения логических устройств приняты следующие обозначения: *a, b, c,* и т.д. (буквы латинского алфавита). Например буквы *a* и *b,* обозначает устройство ввода с дискет (floopy disc). Для указания на то, что это именно устройство, к букве добавляют двоеточие.

Корневой каталог, или каталог, содержащий все другие каталоги и файла, обозначается обратной косой чертой. Все остальные каталоги должны иметь обязательно имя. Например,

Для указания некоторого каталога на данном устройстве необходимо перечислить список каталогов от корневого до данного. Например,

Можно выделить следующие основные файловые операции:

1) открыть файл;
2) закрыть файл;
3) питать данные из файла;
4) писать данные в файл;
5) перемещать указатель файла;
6) проверить конец файла.

Рассмотрим подробнее каждую операцию.

Операция «Открыть файл» обеспечивает:

1) поиск существующего файла или создание нового, в соответствии с указанным путем;
2) проверку прав доступа, например, пользователь может иметь право только читать файл, поэтому открытие на запись в этом случае не будет разрешено;
3) распределение памяти под буфера ввода/вывода это объясняется тем, что физический ввод /вывод осуществляется блоками памяти фиксированной длины, поэтому вместо реальной передачи в файл, производится запись во внутренний буфер, реальную запись в файл производит специальная программа операционной системы, называемая драйвером устройства ввода/вывода;
4) распределение и заполнение специальной структуры, называемой блок управления файлом, в которую записывается: атрибуты файла (размер, время и дата создания, и т.д.), физические адреса файла, указатель файла и др.

Указатель файла специальная переменная блока управления файлом, значение которой указывает на номер (адрес) байта начиная с которого производится операция чтения или записи. После выполнения операции чтения/записи указатель перемещается на количество байт указанных при выполнении операции. Например, указатель файла после выполнении операции «открыть» равен 0, после выполнении операции чтения десяти байт равен 10, если еще прочитаем 20 байт — станет равным 30. Существует специальная операция перемещения указателя файла в заданную позицию, при этом существует три варианта точки опсчета: от начала файла, от конца файла и от текущего значения указателя.

Операция «Писать данные в файл» обеспечивает передачу данных из блока памяти программы во внутренний буфер. Реальную запись в файл осуществляет драйвер, который отслеживает накопление и манипулирование данными. Запись производится в соответствии с указателем файла. После выполнения операции указатель файла перемещается вперед на количество записанных байт.

Операция «Читать данные из файла» обеспечивает передачу данных из файла в блок памяти программы. Реально чтение из файла производится следующим образом: драйвер определяет есть запрашиваемые данные во внутреннем буфере, если есть передает, то передает в блок памяти пользовательской программы, если данных нет, то читает вычисленный на основании указателя файла физический блок памяти, во внутренний буфер, а затем производит передачу в блок программы. Чтение производится в соответствии с указателем файла. После выполнения операции указатель файла перемещается вперед на количество прочитанных байт.

Операция «Закрыть файл» обеспечивает конечную передачу данных из внутренних буферов в файл, редактирует атрибуты файла (например, записывает новый размер файла), освобождает память занимаемые буферами и блок управления файла.

Тестовые операции предназначены для проверки некоторых условий для работы с файлом. Например, достигли конца файла, или имеется данный файл в каталоге и прочее.

Имеется несколько уровней библиотек для работы с файлами. Самый низкий — это уровень операционной системы, как правило, это низкоуровневой программный интерфейс, например, программные прерывания DOS, или функции API Windows и т.д. Интерфейс среднего уровня в Си описывается в заголовочном файле <io.h>, ниже будет описано основное множество

функций этого интерфейса. Здесь опишем интерфейс потокового ввода библиотеки стандартного ввода вывода <stdio.h>.

6.2 Файловая библиотека stdio.h

Для того чтобы использовать функции этой библиотеки необходимо включить заголовочный ***stdio.h***:

#include <stdio.h>

1. Блок описания файла.

Все файловые функции используют специальную структуру FILE описанную в stdio.h

```
typedef struct
{
      unsigned char  *curp;       /* указатель на текущую позицию в буфере
*/
      unsigned char  *buffer;     /* буфер       */
      int            level;       /* уровень буферизации */
      int            bsize;       /* размер буфера              */
      unsigned short  istemp;     /*     */
      unsigned short  flags;      /* флажки управления статуса      */
      wchar_t        hold;        /* текущий символ ели файл открыт без бу-
фера   */
      char           fd;          /* дескриптор файла      */
      unsigned char   token;      /* */
}     FILE;                       /* описание структуры для работы с файлом
*/
```

2. Открыть файл

```
FILE *fopen(const char *filename, const char *mode);
```

fopen открывает файл с именем filename.

Строковый параметр mode используется для указании моды открытия файла и имеет следующие значения:

R	Открыть только для чтения
W	Открыть новый файл, если файл уже существует, то его содержи-

	мое удаляется на его место записывается новый
A	Открыть для записи в конец файла, если файл не существует, то он создается.
r+	Открыть файл для чтения и перезаписи, также можно дописывать в конец файла
w+	Открыть новый файл для чтения и перезаписи, также можно дописывать в конец файла. Если файл существует то старый удаляется
a+	Открыть для записи и перезаписи за концом существующего файла, если файл не существует создается новый.

Возвращаемое значение

При успешном завершении открытия файла fopen возвращает значение указателя на структуру типа FILE, в противном случае возвращает значение NULL.

6.3 Текстовые файлы

Понятие текстового файла появилось на заре компьютерной эры, когда тексты в файле хранились в ASCII коде и многие текстовые редакторы хранили тексты в таких файлах. Особую роль в таких файлах имеет служебные символа «перевод строки» и «возврат каретки». Эти два символа, как правило, указывали на конец строки. Поэтому для работы с такими файлами был специально предусмотрен режим «работа с текстом». В этом режиме некоторые функции автоматически читают строку до указанных служебных символов, не включая сами символы в выходной буфер. Ниже показа программа создающая текстовый файл, записывающая три строки с помощью одного вызова функции fwrite, функция fpus вызывается также один раз и производит запись двух строк. Затем указатель файла перемещается в начало с помощью вызова функции fseek. Далее организуется цикла пока не достигли конца файла и читаем строки с помощью функции fgets. Заодно считаем количество прочитанных строк. Очевидно, их будет 5. Поскольку в файле записано 4 символа возврата каретки (это '\n') и конец файла.

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
int main(void)
{
```

```
  FILE *stream;
  char string[] = "First\nSecond\nThird\n";
  char string2[] = "Hello!\nWorld!!!";
  char msg[20];
  char *ptr;
  int i=0;
  stream = fopen("Example1.txt", "w+t"); //открыть файл в текстовой моде
  fwrite(string, strlen(string), 1, stream); //записать первую строку
  fputs(string2,stream);                     //записать вторую строку
  fseek(stream, 0, SEEK_SET);                //переместить указатель в начало
файла

  while(!feof(stream)){  //цикл для чтения пока не достигли конца файла
    i++;                 //счетчик обращений к файлу
    ptr=fgets(msg, strlen(string)+1, stream); //прочитать строку из тестово-
го файла
    if(ptr!=NULL) printf("%d %s",i, msg);  //отпечатать если прочитана
строка
  }
  fclose(stream);  //закрыть файл
  getch();
  return 0;
}
```

Необходимо отметить, что современные текстовые редакторы имеют собственный формат хранения текста, например, MicrosoftWord имеет формат DOC. В настоящее время файлы содержащие тексты в ASCII коде имеют расшиоение TXT.

6.4 Двоичные файлы

Открыть файл как двоичный (binary) означает, что из файла можно прочитать и записать любую информацию — числа, строки, битовые поля и т.д. Для того чтобы открыть файл как двоичный в функции fopen необходимо указать двоичную моду (суффикс b). Далее, используя, функции fwrite и fread производится запись и чтение данных. Ниже показан пример программы, которая открывает файл как двоичный на чтение и запись. Записывает в него размерность вектора вещественных чисел. Затем записывает сам вектор в файл. Далее указатель файла перемещается в начало файла и заново читается размерность вектора уже из файла. Затем выделяется память заданного размера, читается вектор в новый буфер, затем новый вектор печа-

тается. После печати выделенная динамическая память возвращается, файл закрывается. Программа завершает свою работу.

В этом примере сочетается две операции, как правило, разделенных. Первая это создание двоичного файла. Вторая это чтение данных из файла. Структура файла — первые четыре байта хранят размерность массива. Затем хранятся сами элементы массива. При чтении массива из файла используется механизм распределения памяти. Первоначально читается размерность массива, затем распределяется память, производится чтение массива в динамический буфер.

Ниже приведен исходный текст программы.

```
#include <string.h>
#include <stdio.h>
#include <conio.h>
#include <alloc.h>

int main(void){
  FILE *stream;
  int n=10;
  int m;
  double *z;
  double vec[10]= { 200.0, 100.9, 123.11, 321.33, 456.77, 777.0, 81.98, 990.45,
445.07, 0.009 };
  stream = fopen("Example2.bin", "w+b");  //открыть файл в двоичной моде
  fwrite(&n,sizeof(int) , 1, stream);     //записать размерность
  fwrite(vec,sizeof(double)*10 , 1, stream); //записать

  fseek(stream, 0, SEEK_SET);              //переместить указатель в начало
файла
  fread(&m,sizeof(int) , 1, stream);       //читать размерность
  z=(double*)malloc(sizeof(double)*m);     //выделить память под массив
  fread(z,sizeof(double)*m , 1, stream);   //читать массив чисел
  printf("%d\n",m);
  for(int i=0; i<m; i++)                   //печать массива
   printf("%3.3f ",z[i]);
  printf("\n");                            //освободить память
  free(z);

  fclose(stream);                          //закрыть файл
  getch();
  return 0;
```

```
}
```

Упражнения

1. Написать программу, сохраняющую в файл матрицу вещественных чисел переменной размерности.
2. Написать программу чтения матрицы переменной размерности.

6.5 Библиотеки

Предположим, что в файле необходимо хранить однотипные объекты переменной длины. Например, некоторое множество матриц или изображений. В этом случае поступают следующим образом: каждому информационному объекту присваивается некоторый идентификатор (это может быть имя или некоторый номер), файл разбивается на две части: заголовочная часть и часть хранения объектов. Заголовочная часть хранит таблицу, в которой записано: идентификатор объекта, размер объекта и указатель на расположение этого объекта в файле. Такая организация называется библиотечной. Основные библиотечные операции:

1) создать библиотечный файл;
2) записать объект в библиотеку с заданным идентификатором;
3) прочитать объект по указанному идентификатору;
4) удалить объект из библиотеки.

Рассмотрим более подробнее создание и использование библиотечной организации файла.

Прежде всего, запишем описание заголовка библиотеки. Структура загалока представлена ниже:

```
// описание заголовка библиотеки
typedef struct LibraryHeaderTag {
  char type[10];  //тип файла
  char date[10];  //дата создания
  int  version;   //версия
  int size;       //количество объектов
  int maxsize;    //максимальное количество объектов
  int size_del;   //количество удаленных
  int sizelib;    //размер библиотеки
}  LibraryHeader;   //Имя нового типа
```

Тип файла необходим для идентификации файла, с которым работает программа. Поскольку по ошибке может быть указан файл с другой структурой и это может привести нежелательным последствиям.

Программа в процессе эксплуатации может меняться, добавляются новые элементы в структуру файла или изменяется его структура и т.д., то необходимо устанавливать некоторый идентификатор версии структуры файла.

Параметр size определяет количество объектов, хранящихся на данный момент в библиотеки.

Параметр maxsize определяет максимальное количество объектов которое может храниться в библиотеке.

Параметр size_del определяет количество удаленных объектов. Это необходимо для учета неиспользуемой памяти в библиотеки.

Параметр size_lib хранит текущий размер библиотеки в байтах. Это необходимо для того, чтобы знать адрес записи нового объекта в библиотеку.

Рассмотрим теперь организацию таблицы имен объектов. Описание элемента таблицы приведено ниже:

```
// описание объекта хранения в таблице
typedef struct LibraryElementTag{
  public:
  char name[50]; //имя объекта
  int size;      //размер объекта
  int offset;    //смещение относительно начала файла
  int type;      //тип объекта
} LibraryElement;
```

Имя объекта это любая строка символов идентифицирующая объект в библиотеки, кроме того с именем объекта связывается размер объекта, расположение объекта относительно начала файла, тип объекта необходим, когда в библиотеку записываются разнотипные объекты.

Блок управления библиотекой — структура, которая необходима для работы библиотечных функций. Описание блока управления приведено ниже.

```
//описание блока упраления
typedef struct LibraryControlBlock {
 FILE *hFile;           //дискриптор файла
```

```
 LibraryHeader lh;       //структура заголовка
 LibraryElement *le;     //таблица объектов хранения
} LBC;
```

В этой структуре приведено указатель на структуру FILE для работы с файлом библиотеки, структура описывающая заголовок библиотеки, указатель на таблицу имен объектов. Перечислим основные функции

1. void InitHeader(LibraryHeader *lh,int maxsize);
2. int CreateLibrary(char *name,int maxsize);
3. int OpenLibrary(char *namelib,LBC *lbc);
4. int FindObject(char *name,LBC *lbc);
5. int CloseLibrary(LBC *lbc);
6. int AddLibrary(LBC *lbc,char *name, void* object, int sizeob);
7. int GetSizeObject(LBC *lbc,char *name);
8. int GetObject(LBC *lbc,char *name,void *object).

Функция InitHeader производит инициализацию структуры заголовка библиотеки. Устанавливается тип библиотеки, дата создания, версия, максимальное количество объектов, первоначальное значение размера библиотеки. Ниже приведен текст программы

```
void InitHeader(LibraryHeader *lh,int maxsize){
 strcpy(lh->type,"krulib");
 strcpy(lh->date,"7.01.2004");
 lh->version=1;
 lh->size=0;
 lh->maxsize=maxsize;
 lh->size_del=0;
 lh->sizelib=sizeof(LibraryHeader)+maxsize*sizeof(LibraryElement);
}
```

Функция создания библиотечного файла CreateLibrary, создается файл с заданным именем name. Далее инициализируется заголовочная структура и записывается в файл, затем организуется цикл и записывается множество пустых элементов таблицы количеством maxsize.

```
int CreateLibrary(char *name,int maxsize){
  FILE *hFile;
  LibraryHeader lh;
  LibraryElement le;
```

```
  hFile=fopen(name,"a+"); //открыть файл несуществующий
  if(hFile==NULL) return -1; //обработка ошибки
  InitHeader(&lh,maxsize);   //инициализировать заголовок
  fwrite(&lh,1,sizeof(LibraryHeader),hFile);  //записать заголовок
   setmem(&le,sizeof(LibraryElement),0);         //обнулить структуру элемента
таблицы
  for(int i=0; i<maxsize; i++)            //записать заданое максимальное количе-
ство элементов
   fwrite(&le,1,sizeof(LibraryElement),hFile);
   fclose(hFile);  //закрыть файл
  return 1;
}
```

Функция открытия или создания библиотечного файла OpenFile. Первоначально вызывается библиотечная функция access, описание которой дано в заголовочном файле io.h. Эта функция проверяет наличие файла в текущем каталоге и если доступ к файлу имеется, то возвращает 0, в противном случае, код ошибки. В нашем случае возврат нуля означает присутствие файла в каталоге. Если файл не существует, то вызывается функция CreateLibrary. Если файл существует то функция открывает файл с именем указанном в namelib. Далее читается заголовок и проверяется тип файла. Если тип совпал, то проверяется версия, и далее распределяется память под таблицы, если таблица не пуста, то читается таблица. По окончанию открытия возвращается код завершения и заполненная структура блока управления библиотекой.

```
int OpenLibrary(char *namelib,LBC *lbc){
  if(access(namelib, 0) != 0){  //файл с таким именем не существует?
   if(CreateLibrary(namelib,100)==-1) return -1;     //да - создать!
   }
  lbc->hFile=fopen(namelib,"r+b");  //открыть на чтение и перезапись с добав-
лением
  if(lbc->hFile==NULL) return -1;  //обработка ошибки
  fseek(lbc->hFile,0L,SEEK_SET);   //переместить указатель файла в начало
  fread(&lbc->lh,1,sizeof(LibraryHeader),lbc->hFile); //прочитать заголовок
  if(strcmp(lbc->lh.type,"krulib")!=0) { //проверка типа файла
    printf("Error %s type\n",namelib);
    return -2;
   }
  if(lbc->lh.version!=1) { //проверка версии файла
    printf("Error %s version\n",namelib);
```

```
   return -3;
  }
 //распределить память под таблицу
 lbc->le=(LibraryElement *)malloc(sizeof(LibraryElement)*lbc->lh.maxsize);
 if(lbc->le==NULL) return -1;
 if(lbc->lh.size>0) //если таблица не пуста, прочитать таблицу
  fread(lbc->le,1,sizeof(LibraryElement)*lbc->lh.size,lbc->hFile);
 return 1;
}
```

Функция закрытия библиотеки CloseLibrary выполняет следующие действия:

1. Переписывается заголовок библиотеки из блока управления.
2. Переписывается таблица имен элементов.
3. Закрывается файл библиотеки.

Структура lbc должна быть актуальной.

```
int CloseLibrary(LBC *lbc){
 fseek(lbc->hFile,0L,SEEK_SET);
 fwrite(&lbc->lh,1,sizeof(LibraryHeader),lbc->hFile);
 if(lbc->lh.size>0)
  fwrite(lbc->le,1,sizeof(LibraryElement)*lbc->lh.size,lbc->hFile);
 fclose(lbc->hFile);
 free(lbc->le);
 return 1;
}
```

Функция FindObject ищет имя в таблице имен, которая размещена в блоке управления библиотекой. Если такое имя находит, то возвращает индекс. В противном случае — 1.

```
int FindObject(char *name,LBC *lbc){
   for(int i=0; i<lbc->lh.size; i++){
    if(strcmp(name,lbc->le[i].name)==0) return i; //нашли
   }
   return -1; //не нашли
 }
```

Функция AddLibrary производит запись объекта в библиотеку. Предварительно структура lbc должна быть заполнена функцией OpenLibrary. Кроме того, необходимо передать имя объекта, его адрес и размер. Функция проверяет наличие такого имени в библиотеки, проверяет наличие памяти для записи в таблицу имен и если все нормально, то записывает имя,

размер и адрес (местоположение объекта уже в файле) в заданный элемент таблицы. Далее записывается объект в конец файла. Затем корректируется в блоке управления размер файла.

```
int AddLibrary(LBC *lbc,char *name, void* object, int sizeob){
  if(FindObject(name,lbc)==-1){ //поискать объект с таким именем
   if(lbc->lh.size==lbc->lh.maxsize) return -1; //место еще есть?
   fseek(lbc->hFile,0L,SEEK_END); //переместить указатель файла в конец
   int i=lbc->lh.size++;          //увеличить счетчик числа объектов
   strcpy(lbc->le[i].name,name);  //записать имя в таблицу
   lbc->le[i].size=sizeob;        //записать размер объекта
    lbc->le[i].offset=lbc->lh.sizelib; //записать смещение относительно начала
файла
   fwrite((char*)object,1,sizeob,lbc->hFile); //записать объект в конец файла
   lbc->lh.sizelib+=sizeob;   //изменить значения размера файла
   return 1;
  }
  else return 0;  //объект с таким именем существует
}
```

Функция GetSizeObject определяет размер объект.

```
int GetSizeObject(LBC *lbc,char *name){
  int addr;
  if((addr=FindObject(name,lbc))==-1) return 0; //не нашли
  return lbc->le[addr].size;
}
```

Функция GetObject читает объект по имени.

```
int GetObject(LBC *lbc,char *name,void *object){
  int addr;
  if((addr=FindObject(name,lbc))==0) return 0; //не нашли
  //переместить указатель файла на смещение указанное в элементе таблицы
(addr)
  fseek(lbc->hFile,lbc->le[addr].offset,SEEK_SET);
  //прочитать содержимое объекта
  fread((char*)object,1,lbc->le[addr].size,lbc->hFile);
  return 1;
  }
```

Пример использования описанных функций

```
void main(){
 LBC lbc;  //объявляем блок управления библиотекой
 OpenLibrary("xxx.lib",&lbc);  //создаем или открываем
 char nameo[10]="name0";
 char object[100]="object0";
 //for(int i=0; i<10; i++){
 //  nameo[4]='a'+i;
 //  object[6]='0'+i;
 //  AddLibrary(&lbc,nameo,object,strlen(object));
 //}
 int size;
 for(int i=0; i<10; i++){
   nameo[4]='a'+i;
   object[6]='0'+i;
   size=GetSizeObject(&lbc,nameo);
   GetObject(&lbc,nameo,object);
   object[size]='\0';
   printf("%s\n",object);
 }
 CloseLibrary(&lbc);
 getch();
}
```

Задания:

1) перезапись объекта в библиотеке;
2) сборка мусора;
3) объединение двух библиотек;
4) реорганизация библиотеки;
5) списочное представление объектов библиотеки;
6) понятие архива (сжатие и кодирование).

6.6 Базы данных

6.6.1 Общие сведения о базах данных

Базы данных еще одно из фундаментальных видов систем программирования. Как известно [ ], первые ЭВМ применялись для инженерных и научных расчетов, однако разработка и внедрение устройств долговременной памяти и большой емкостью (магнитные ленты, барабаны, диски) дали

толчок к использованию ЭВМ для хранения и обработки больших массивов информации. Как правило, это экономическая информация, которая характеризуется большим объемом однотипной структуры и простыми алгоритмами обработки. Например, информация о служащих фирмы или банка, информация о закупке сырья и выпуска товара для производственных фирм и т.д.

Развитие этого направления привело к понятию систем управления базами данных (СУБД). Эти системы обеспечивают некоторые универсальные средства для хранения и манипулирования данными. Как правило, СУБД — это инструментальная система программирования для создания корпоративных информационных систем. В основе которой лежит специальный алгоритмический язык. Боле подробное описание можно найти в литературе [ ]

В настоящее время имеется три основных модели, лежащих в основе построения баз данных: иерархические, сетевые и реляционные [ ]. Для исследования последней модели разработан специальный математический аппарат-реляционная алгебра [ ]. В настоящее время практически все СУБД основаны на реляционной модели представления данных.

В основе всех современных СУБД лежат обычные файловые операции, такие как отрыть файл, закрыть файл, прочитать данные, записать данные и проч. Физически базы данных представляются совокупностью логически связанных файлов. Как правило множество файлов делится на два типа, в файлах первого типа хранятся данные, файлы второго типа являются служебными и предназначены для организации эффективного поиска и обработки данных, как правило, эти файлы называются индексными.

Ниже предлагается простой пример базы данных, которой показывает основные идеи и механизмы, заложенные в современных базах данных.

6.6.2 Библиотека файловых функций io.h

Весь пример базируется на использовании библиотеки файловых функций среднего уровня Си, описание этих функций дано в заголовочном файле io.h. Основные функции этой библиотеки следующие:

1) open — открыть файл;
2) close — закрыть файл;
3) read — читать данные из файла;
4) write — писать данные в файл;
5) lseek — переместить указатель файла в заданную позицию.

Эти функции похожи на функции, описанные в заголовочном файле stdio.h.

6.6.3 Физическая организация файла данных

В отличии от библиотечной структуры, дынные которые записываются в файл имеют фиксированный размер и называются записью. Запись обычно представляется некоторой структурой, в которой записан тип и размер каждого поля данных. Например, описание информации о книге в студенческой библиотеке.

```
// описание записи
typedef struct MyRecord {
 char sdel;              //признак удаления записи
 char Name[40];          //имя автора
 char Caption[255];      //название
 char Year[20];          //год издания
 int  Pages;             //количество страниц
 int  Nal;               //сколько имеется
} RECORD;
```

Хотя структура записи для функций работающих с файлом данных не важна. Важно то, что размер записи фиксирован. Итак физическая организация файла данных организована следующим образом:

1) имеется заголовок файла данных, в котором записана обобщенная информация о файле данных, этот заголовок записан в начале файла и имеется механизм определения размера этого заголовка;
2) после заголовка следуют записи описанной выше структуры, поскольку записи фиксированной длины, к каждую запись можно идентифицировать по ее номеру, начиная с нулевого. Нулевая запись следует сразу после заголовка, первая — после нулевой и т.д. Номер записи называется индексом записи.

Итак, к данным можно добрать зная индекс записи. Формула вычисления адреса записи по ее индексу следующая:

Адрес_записи=размер_заголовка+индекс*размер_записи

Рассмотрим структура заголовка файла данных:

```
typedef struct MyBaseContBlock   {
   int n_rec;        //количество записей
   int n_del;        //количество удаленных записей
   int hand;         //дескриптор файла
   int sizeBCB;      //размер блока
```

```
  int sizeRecord;     //длина записи
  long sizeBase;      //длина файла (базы данных)
 } BCB;
```

Как видно из описания хранится общее количество записей, количество удаленных записей, размер заголовка, размер записи, размер файла. Может также храниться и другая информация, например, владелец, описание структуры записи, версия, тип и т.д.

6.6.4 Основные функции для работы с файлом данных

Основные функции следующие:

1) открыть или создать файл данных OpenBase;
2) закрыть файл данных CloseBase;
3) добавить новую запись в конец файла AddRecord;
4) прочитать запись по индексу ReadRecord;
5) переписать имеющуюся запись на новую RewriteRecord;
6) логически удалить запись DeleteRecord;
7) сборка мусора Sborka.

Функция открыть или создать файл данных проверяет наличие файла данных с таки именем, если нет то создает заголовок и записывает его в новый файл. Если такой файл существует, то открывает его и читает заголовок файла данных и заполняет структуру BCB. Далее все остальные функции могут работать с файлом, используя структуру BCB.

Значения флажков в функции open следующие: первая группа описывает операции, вторая — права на выполнение операций чтения и записи. Группа вторых флажков может отсуствовать.

```
int OpenBase(char* NameBase,BCB *b){
 int fm;
 if(access(NameBase, 0) != 0){  //файл с таким именем не существует?
  Init(b);                      //да - создать
  fm=open(NameBase,O_CREAT|O_RDWR|O_BINARY,S_IREAD|
S_IWRITE);
  if(fm==-1){ //обработка ошибки
   printf("Error open base: errno- %d\n",errno);
   return -1;
   }
  b->hand=fm;                   //установить дескриптор
  write(fm,b,sizeof(BCB)); //записать блок управления
  PrintBCB(b);                  //печать блока управления
```

```
 }
 else { //база данных существует
  fm=open(NameBase,O_CREAT|O_RDWR|O_BINARY);
 if(fm==-1){  //обработка ошибки
   printf("Error open base: errno- %d\n",errno);
   return -1;
 }
  read(fm,b,sizeof(*b));  //читать заголовок
  b->hand=fm;              //установить дескриптор
  PrintBCB(b);             //печать блока управления
   }
  return 1;
}
```

Функция закрыть базу данных производит запись заголовка в файл и закрытие файла данных с помощью функции close.

```
void CloseBase(BCB *b){
  lseek(b->hand,0L,SEEK_SET);   //сбросить указатель файла в начало
  write(b->hand,b,sizeof(BCB));    //записать новое значение бока управления
  close(b->hand);                  //закрыть файл
  b->hand=-1;                      //сбросить дескриптор файла
}
```

Функция добавить новую запись в конец файла перемещает указатель в конец файла, и производить добавление новой записи. При этом длинна файла увеличивается фиксированный размер и увеличивается значение счетчика записей в блоке управления файлом BCB.

```
void AddRecord(BCB *b,RECORD *rec){
 lseek(b->hand,0L,SEEK_END);       //указатель файла переместить в конец
 write(b->hand,rec,b->sizeRecord);    //произвести запись
 b->n_rec++;                          //увеличить счетчик записей
}
```

Функция прочитать запись по индексу производит вычисление и перемещение указателя файла в нужную позицию (описание формулы смотри выше). Далее производится чтение записи в указанный буфер.

```
void ReadRecord(BCB *b,int addr,RECORD *rec){
  //вычислить и переместить указатель в файла на заданную запись
```

```
 lseek(b->hand,(b->sizeBCB+addr*b->sizeRecord),SEEK_SET);
 read(b->hand,rec,b->sizeRecord);                //читать запись
}
```

Функция переписать заданную запись производит изменение старой записи в файле данных.

По заданному индексу ищется позиция записи в файле и туда перемещается указатель файла. Дальше производится перезапись информации из буфера rec в файл.

```
void RewriteRecord(BCB *b,int addr,RECORD *rec){
 //вычислить и переместить указатель в файла на заданную запись
 lseek(b->hand,(b->sizeBCB+addr*b->sizeRecord),SEEK_SET);
 write(b->hand,rec,b->sizeRecord);   //произвести новую запись
}
```

Функция логического удаление записи производится следующим образом. Запись физически из файла не удаляется, а просто помечается как удаленная. Для этого предусмотрен специальный байт удаления. В нашем случае это первый байт записи. Наличие символа звездочки говорит о том что запись удалена.

```
void DeleteRecord(BCB *b,int addr){
 char sdel='*';
 if(addr>=b->n_rec) return; //неверный номер записи
 //вычислить и переместить указатель в файла на заданную запись
 lseek(b->hand,(b->sizeBCB+addr*b->sizeRecord),SEEK_SET);
 write(b->hand,&sdel,1);   //произвести логическое удаление записи
 b->n_del++;
}
```

Функция сборки мусора производит физическое удаление записей из файла данных.

Реорганизацию файла данных можно осуществить двумя путями: Перезапись всех не удаленных записей данных в текущем файле данных (упаковка) и второй это создание нового файла и запись в новый всех действительных записей. После выполнения операции старый файл удаляется, а новый переименуется. Второй вариант более надежен, т.к. при сбое в первом случае можно потерять информацию, во втором случае информация не теряется, т.к. старый удаляется после создания нового. Ниже представлен текст функции сборки мусора по второму варианту. При этом использует-

ся две стандартные функции. Функция unlink производит удаление указанного файла, описана в io.h. Вторая функция rename переименовывает файл с именем, указанном в первом аргументе, на имя указанное во втором аргументе, описана в stdio.h.

```
void Sborka(BCB *b){
  BCB nb;
  RECORD rec;
  if(OpenBase("tmp$$$.dat",&nb)==-1) //создать новую базу
    return;

  for(int i=0; i<b->n_rec; i++){ //прочитать все записи в файле данных
   ReadRecord(b,i,&rec);
   if(rec.sdel!='*')  //эта запись удалена?
      AddRecord(&nb,&rec);  //нет, записываем в новую.
   }
  CloseBase(&nb);  //закрыть новую базу
  CloseBase(b);    //закрыть старую базу
  unlink("mybook.dat");  //удалить старую
  rename("tmp$$$.dat","mybook.dat"); //переименовать новую в старую
}

//Удалить базу
void DeleteBase(){
 if(unlink("mybook.dat"))  //удалить
  printf("Error open base: errno- %d\n",errno);
}

//печать блока управления
void PrintBCB(BCB *b){
  printf("BCB: number of record %d\n",b->n_rec);
  printf("BCB: number of delete record %d\n",b->n_del);
  printf("BCB: size of BCB: %d\n",b->sizeBCB);
  printf("BCB: size of record: %d\n",b->sizeRecord);
  printf("BCB: size of base: %d\n",b->sizeBase);
}

//инициализация блока управления
void Init(BCB *b){
 b->n_rec=0;
 b->n_del=0;
```

```
 b->sizeBCB=sizeof(BCB);
 b->sizeRecord=sizeof(RECORD);
 b->sizeBase=200000;
}

#include <stdio.h>
#include <io.h>
#include <fcntl.h>
#include <string.h>
#include <sys\stat.h>
#include <errno.h>
#include <conio.h>
#include <stdlib.h>
#include <mem.h>
// описание блока управления базой данных


//печать записи
void PrintRecord(RECORD *r){
 printf("%c %s %s %s %d %d\n",r->sdel,r->Name,r->Caption,r->Year,r-
>Pages,r->Nal);
}

//Сгенерировать и записать nrec записей в базу данных
void CreateBase(int nrec,BCB *b){
 RECORD r;
 int i, k;
 char bu[10];
 randomize();  //используется датчик случайных чисел
 for(i=0; i<nrec; i++){
  k=random(100); //получить номер автора
  sprintf(bu,"%d",k);
  strcpy(r.Name,"Author");
  strcat(r.Name,bu);
  k=random(100);     //получить номер книги
  sprintf(bu,"%d",k);
  strcpy(r.Caption,"Book");
  strcat(r.Caption,bu);
  k=1900+random(104);  //получить год издания
  sprintf(bu,"%d",k);
  strcpy(r.Year,bu);
```

```
  r.Pages=100+random(100);  //получить количество страниц
  r.Nal=1+random(20);       //получить количество имеющихся книг
  AddRecord(b,&r);
 }
}
```

6.6.5 Индексные файлы

Поиск и выборка информации эффективны по упорядоченным данным. Упорядочение некоторой последовательности данных называется сортировкой. Поскольку запись хранит несколько различных полей, которым может быть произведена сортировка, со сам файл данных также может отсортирован несколькими вариантами. Однако необходимо признать что сортировка файла данных является достаточно длительной операцией. Поэтому поступают следующим образом: сам файл данных не сортируется. Вместо этого для каждого вида сортировки создается файл, в котором хранятся в отсортированном порядке не записи, а их индексы (номер записи в файле данных). Поэтому такие файлы называются индексными.

Индексным файлом называется файл, содержащий индексы записей файла данных в соответствии с некоторым критерием сортировки. Таким образом, для одного файла данных может быть несколько индексных файлов.

Рассмотрим построение индексного файла для нашего примера. Индексный файл создается на основе сортировки файла данных по какому либо критерию. Поэтому для создания индексного файла необходимо задать критерий сортировки (или ключ сортировки). В нашем случаем в качестве ключа выбрано поле Name в структуре записи (сортировка по имени автора книги).

Функция создания индексного файла CreateIndex. Это упрощенный пример сортировки фала данных и создание упорядоченного индексного файла. Первоначально распределяется память под массив индексов и он инициализируется. Далее он упорядочивается в соответствии с алгоритмом лексикографического упорядочения (см. раздел Сортировка строк). После сортировки массив записывается в файл.

```
void CreateIndex(BCB *b){
 int key=1;
 int tmp;
 RECORD rec1, rec2;
 int *index=(int*)malloc(sizeof(int)*b->n_rec);
 for(int i=0; i<b->n_rec; i++) index[i]=i;
```

```
 while(key){
  key=0;
  ReadRecord(b,index[0],&rec1);
  for(int i=1; i<b->n_rec; i++){
   ReadRecord(b,index[i],&rec2);
   if(strcmp(rec1.Name,rec2.Name)>0){
    tmp=index[i-1];
    index[i-1]=index[i];
    index[i]=tmp;
    key=1;
   }
  else
   memcpy(&rec1,&rec2,sizeof(RECORD));
  }
 }//end while
  int fm=open("mybook.inx",O_CREAT|O_RDWR|O_BINARY,S_IREAD|
S_IWRITE);
  if(fm==-1){ //обработка ошибки
   printf("Error open base: errno- %d\n",errno);
   return;
   }
 write(fm,index,sizeof(int)*b->n_rec); //записать индексы
 close(fm);
}
```

Функция ViewIndex предназначена для выдачи всего файла данных в соответствии следования индексов в индексном файле. Используется функции ReadRecord и PrintRecord

```
void ViewIndex(BCB *b){
 RECORD rec;
 int *index=(int*)malloc(sizeof(int)*b->n_rec); //распределяем память
//открываем файл
 int fm=open("mybook.inx",O_CREAT|O_RDWR|O_BINARY,S_IREAD|
S_IWRITE);
  if(fm==-1){ //обработка ошибки
   printf("Error open base: errno- %d\n",errno);
   return;
   }
 read(fm,index,sizeof(int)*b->n_rec); //читать индексы
 close(fm);
```

```
 for(int i=0; i<b->n_rec; i++){
  ReadRecord(b,index[i],&rec);
  printf("%d index=%d rec:",i,index[i]);
  PrintRecord(&rec);
 }
}
```

Функция FindName производит быстрый поиск нужной записи по полю Name. Поиск организован следующим образом, задается интервал индексов g1 и g2, первоначально интервал охватывает весь файл индексов. Затем берется середина интервала, читается запись и сравниваются — поле Name и заданная строка, используя функцию strcmp. Далее если строки совпали (равенство нулю), то нужную запись нашли. Если strcmp вернула 1, то

g2=addr (смещение влево относительно середины addr), в противном случае смещение вправо

g1=addr. Поиск будет производиться пока либо не найдется запись, либо интервал не станем равным единице.

```
int FindName(BCB *b,char *Name, RECORD *rec){
 int g1,g2;
 int res, addr, newaddr;
 int *index=(int*)malloc(sizeof(int)*b->n_rec);
 int fm=open("mybook.inx",O_CREAT|O_RDWR|O_BINARY,S_IREAD|
S_IWRITE);
  if(fm==-1){ //обработка ошибки
   printf("Error open base: errno- %d\n",errno);
   return -1;
   }
 read(fm,index,sizeof(int)*b->n_rec); //читать индексы
 close(fm);
 addr=(b->n_rec-1)/2; //середина
 g1=0; g2=b->n_rec-1;
 while(1){
  ReadRecord(b,index[addr],rec);
  printf("%d index=%d rec:",addr,index[addr]);
  PrintRecord(rec);
  if((res=strcmp(rec->Name,Name))==0) return 1;
  else
  if(res>0) g2=addr-1;
```

```
 else      g1=addr+1;
 newaddr=g1+(g2-g1)/2;
 if(addr==newaddr) return 0;
 addr=newaddr;
 }//end while
}
```

Выше была показана простейшая организация файла данных и индексного файла. Очевидно, что эти два файла связаны между собой и все изменения в файле данных должны отражаться в индексных файлах. Поскольку операции вставки, переписи и удаления явление обычное для баз данных, то необходимо иметь эффективные методы актуализации индексных файлов. Обычно для этих целей используются методы организации индексных файлов, базирующихся на B-деревьях. Более подробно об этом можно почитать …

Функции диалога для ввода записи, манипуляции с записями в базе, манипуляции с самой базой данных

```
//ввод записи
void InputRecord(RECORD *r){
 char ch;
 char str[80];
 setmem(r,sizeof(RECORD),0); //очистить буфер записи
 while(1){ //цикл для ввода полей записи
 puts("1-Name; 2-Caption; 3-Year; 4-Pages; 5-Nal; 6-View; esc-Exit");
 ch=getch();
 printf("%c\n",ch);
  switch(ch){
  case '1': printf("Name>"); gets(r->Name);                    break;
  case '2': printf("Caption>"); gets(r->Caption);             break;
  case '3': printf("Year>"); gets(r->Year);                   break;
  case '4': printf("Pages>"); gets(str); r->Pages=atoi(str);    break;
  case '5': printf("Nal>"); gets(str);   r->Nal=atoi(str);      break;
  case '6': PrintRecord(r);                                   break;
  case  27: return;
  }
 }
}
```

Функция для ввода, чтения, перезаписи, добавления, просмотра и удаления записей

```
void WorkRecord(BCB *b){
 char ch;
 RECORD r;
 int addr;
 char str[80];
 while(1){
 puts("----------------------");
 puts(" 1-Input;\n 2-Add;\n 3-Rewrite;\n 4-View;\n 5-Delete;\n ESC- exit");
 printf("record>");
 ch=getch();
 printf("%c\n",ch);
  switch(ch){
  case 27:  return; //esc
  case '1': InputRecord(&r);
            break;
  case '2': AddRecord(b,&r);
            break;
  case '3': printf("Rewrite Addr>");
            gets(str);
            addr=atoi(str);
            RewriteRecord(b,addr,&r);
            break;
  case '4': printf("Read Addr>");
            gets(str);
            addr=atoi(str);
            ReadRecord(b,addr,&r);
            PrintRecord(&r);
            break;
  case '5': printf("Delete Addr>");
            gets(str);
            addr=atoi(str);
            if(addr==-1) break;
            DeleteRecord(b,addr);
            break;

    }
 }
}
```

Функция Work организует простой диалог для работы с базой данных.

```
void Work(){
 char ch;
 BCB b;
 RECORD r;
 int addr;
 char str[80];
 if(OpenBase("mybook.dat",&b)==-1) return;
 while(1){
 puts("----------------------");
 puts(" 1-View;\n 2-Create nrec;\n 3-Record;\n 4-Delete;\n 5-Sborka;\n\
  6-Sort;\n 7-ViewSort;\n 8-FindName;\n esc-exit");
 printf("base>");
 ch=getch();
 printf("%c\n",ch);
  switch(ch){
 case '1': for(int i=0; i<b.n_rec; i++){
             ReadRecord(&b,i,&r);
             printf(" %d ",i);
             PrintRecord(&r);
            }
           break;
  case '2': printf("Create nrec>");
           gets(str);
           addr=atoi(str);
           CreateBase(addr,&b);
           break;
  case '3': WorkRecord(&b);
           break;
  case '4': CloseBase(&b);
           DeleteBase();
           return;
  case '5': Sborka(&b);
           return;
  case '6': Sort(&b);
           return;
  case '7': ViewSort(&b);
           break;
```

```
  case '8': printf("FindName>");
            gets(str);
            if(FindName(&b,str,&r)==1)  PrintRecord(&r);
            else printf("Name %s don't find\n",str);
            break;
  case 27:  CloseBase(&b);
            return; //esc
     }
 }
}
```

7 ДЕРЕВЬЯ, ДРЕВОВИДНЫЕ СПИСКИ

7.1 Основные понятия и определения

Деревья являются удобным способом представления информации. Поэтому в программировании они играют огромную роль. Вместе с тем можно выделить два наиболее важных направления:

1) использование деревьев для представления информации, например древовидная структура каталогов, древовидная структура некоторой фирмы (директор-подразделения-отделы-работкники), иерархические классификации и т.д.;

2) использование деревьев для организации эффективного поиска информации (двоичные деревья, АВЛ-деревья, В-деревья и т.д.).

Дадим ряд определений.

Деревом будем называть ациклический ориентированный граф *G*, обладающий следующими свойствами:

1) имеется узел r, у которого нет входных дуг, назовем его корнем дерева;

2) все узлы, кроме корня, имеют одну входную дугу и несколько выходных дуг (выходные дуги могут отсутствовать).

Узел, у которого нет выходных дуг, назовем листом.

Сыном узла *z* будем называть узел *w*, который имеет входную дугу выходящую из *z*. **Потомком** узла *z* будем считать узел *w*, к которому есть путь от *z* по выходным дугам. **Предком** узла w будем называть узел z, от которого имеется путь по выходным дугам.

Поддеревом дерева G будем называть дерево, получаемое из G путем отсечения некоторого подмножества входных дуг

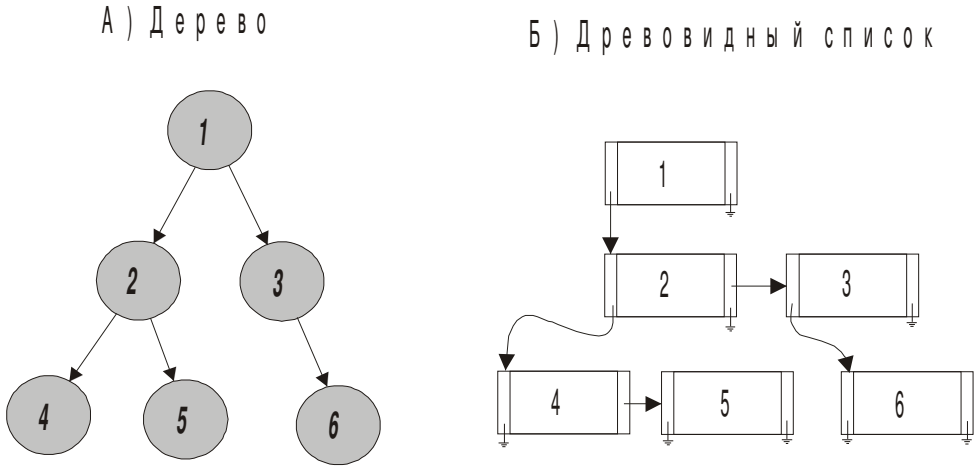


Рис. 7.1

На рис. 7.1 узел под номером 1 — корень дерева, узлы 4, 5, 6 — листья, узлы 4, 5 для узла 2 являются сыновьями, узел 1 для узла 6 является предком, а узлы 4, 5, 6 являются потомками для узла 1.

7.2 Представление деревьев

Скобочная запись представления дерева осуществляется следующим образом: записывается название узла, затем в скобках записываются сыновья. Например, для описание дерева рис 1 :

Корень1 (Узел2(Лист4,Лист5), Узел3(Лист6)).

Нумерация узлов с помощью кода Дьюи, например,

1.Корень
1.1.Узел2
1.1.1.Лист4
1.1.2.Лист5
1.2.Узел2
1.2.1.Лист6

Отсюда видно, что любой учебник можно представить в виде некоторого дерева

Учебник(Введение,Глава1(Параграф1(Раздел1(Пункт1)..)..),Глава2(…), …, Заключение,Литература,Приложения)

7.2.1 Представление дерева в виде древовидного списка

Другим важным представлением дерева в памяти компьютера являются древовидные списки. Каждый узел дерева представлен элементом списка, который содержит, в простейшем случае, два указателя: указатель на список сыновей и указатель на правого брата. Указатели представлены стрелками (рис.). Отсутствие указателей обозначено символом заземления.

Рассмотри реализацию функций для работы с древовидным списком, представленным как контейнер.

Описание элемента древовидного списка. Имеется три указателя первый указатель на список сыновей, второй указатель на брата справа. Третий указатель это указатель на любые данные, которые программист устанавливает по своему усмотрению.

```
typedef struct TagNodeTreeList{
  struct TagNodeTreeList *down;  //указатель на список сыновей
  struct TagNodeTreeList *right;   //указатель на соседа справа
  void* data;                      //данные узла
} NodeTreeList;
```

Структура описывающая весь древовидный список.

```
typedef struct {
  NodeTreeList *root; //корень дерева
  void *discrip;          //описание (используют клиенты)
} KruTreeList;
```

Описание функций для работы с древовидным списком

1. Функция создания древовидного списка. Выделяется память под структуру KruTreeList и устанавливаются начальные значения элементов структуры. Если памяти не было выделено, то возвращается нулевое значение указателя.

```
KruTreeList *CreateTreeList(){  //создание древовидного списка
 KruTreeList *tl=(KruTreeList *)malloc(sizeof(KruTreeList));
 if(tl==NULL) return NULL;
 tl->root=NULL;
 tl->discrip=NULL;
 return tl;
}
```

2. Функция создание узла древовидного списка. На входе подается указатель на данное, который надо записать в элемент древовидного списка. Выделяется память под структуру NodeTreeList, устанавливаются указатели down и right в нулевое значение, указатель на данные записывается в node->data. Возвращается указатель на структуру NodeTreeList.

```
NodeTreeList* CreateTreeListNodeData(void* data){
 NodeTreeList *node=(NodeTreeList*)malloc(sizeof(NodeTreeList));
 if(node==NULL) return NULL;
 node->down=NULL;
 node->right=NULL;
 node->data=data;
 return node;
}
```

3. Функция копирование узла древовидного списка. На входе подается элемент древовидного списка и указатель на функцию, которая выделяет память под новую копию данных и копирует данные в новый буфер. Первоначально выделяется память под узел, устанавливается нулевые значения указателей down и right, если указатель на функцию нулевой, то копируется указатель на данные. В противном случае производится копирование путем вызова функции копирования данных и запись адреса копии в поле cnode->data. Функция возвращает адрес копии узла древовидного списка.

```
NodeTreeList* CopyTreeListNode(NodeTreeList* node,void*(*CopyData)
(void *)){
 NodeTreeList *cnode=(NodeTreeList*)malloc(sizeof(NodeTreeList));
 if(cnode==NULL) return NULL;
 cnode->down=NULL;
 cnode->right=NULL;
 if(CopyData==NULL) cnode->data=node->data;
 else
```

```
 cnode->data=CopyData(node->data);
 return cnode;
}
```

4. Функция копирования поддерева. Функция производит рекурсивный обход древовидного списка и копирует узлы и устанавливается соответствующие связи .

```
NodeTreeList*  CopyTreeListSubTree(NodeTreeList*  node,void*(*CopyData)
(void *)){
 if(node==NULL) return NULL;
 NodeTreeList* cnode=CopyTreeListNode(node,CopyData); //копируем узел
 if(cnode==NULL) return NULL;
     cnode->down=CopyTreeListSubTree(node->down,CopyData);  //копируем
поддерево
  cnode->right=CopyTreeListSubTree(node->right,CopyData); //копируем правого узла
  return cnode;
}
```

5. Функция вставки узла как самого левого сына. На входе задан адрес узла, к которому вставляется новый узел с данными адрес, которых помещен в указатель data.

```
void AddTreeListNodeLeftSonData(NodeTreeList *node,void *data){
 NodeTreeList *son;
 if(node==NULL) return;
 son=CreateTreeListNode(data);
 if(son==NULL) return;
 if(node->down==NULL) //у узла нет сыновей
   node->down=son;
 else { //у узла имеются сыновья
  son->right=node->down;
  node->down=son;
  }
}
```

6. Функция вставки поддерева, корень которого записывается как самый левый сын указанного узла. На входе заданы: узел node и корень поддерева son.

```
void AddTreeListNodeLeftSon(NodeTreeList *node,NodeTreeList *son){
 if(node==NULL||son==NULL) return;
 if(node->down==NULL) //у узла нет сыновей
   node->down=son;
 else { //у узла имеются сыновья
  son->right=node->down;
  node->down=son;
  }
}
```

7. Функция вставки узла как самого правого сына. На входе задан адрес узла, к которому вставляется новый узел с данными адрес, которых помещен в указатель data.

```
void AddTreeListNodeRightSonData(NodeTreeList *node,void *data){
 NodeTreeList *son;
 if(node==NULL) return;
 son=CreateTreeListNode(data); //создаем новый узел
 if(son==NULL) return;
 if(node->down==NULL) //у узла нет сыновей
   node->down=son;
 else { //у узла имеются сыновья
  NodeTreeList *s=node->down;
  while(s->right!=NULL) s=s->right; //находим самого правого сына
  s->right=son;
  }
}
```

8. Функция вставки поддерева, корень которого записывается как самый правый сын указанного узла. На входе заданы: узел node и корень поддерева son.

```
void AddTreeListNodeRightSon(NodeTreeList *node,NodeTreeList *son){
 if(node==NULL||son==NULL) return;
 if(node->down==NULL) //у узла нет сыновей
   node->down=son;
 else { //у узла имеются сыновья
  NodeTreeList *s=node->down;
  while(s->right!=NULL) s=s->right; //находим самого правого сына
  s->right=son;
  }
}
```

9. Функция удаления поддерева узла. На входе указывается указатель на корень поддерева и указатель на функцию удаления данных DeleteData. Производится рекурсивный обход поддерева и удаление узлов вместе с данными.

```
void DeleteSubTreeList(NodeTreeList *node,void(*DeleteData)(void*)){
  if(node==NULL) return;
  DeleteSubTreeList(node->down,DeleteData); //удалить левого сына
  DeleteSubTreeList(node->right,DeleteData); //удалить правого брата
  DeleteData(node->data); //удалить данные узла
  free(node); //удалить узел
}
```

10. Функция удаления сына вместе с поддеревом. Первоначально удаляется поддерево, затем корректируются связи, в зависимости от расположения узла и удаляется сам узел.

```
void DeleteSonTreeList(NodeTreeList *node,NodeTreeList *son,void(*Delete-
Data)(void*)){
  if(node==NULL) DeleteSubTreeList(son, DeleteData); //удаление всего дере-
ва
 else{  //найти  соседей
   if(node->down==son) //самый левый сын
    node->down=son->right; //у родителя изменяем связь на левого сына
   else{         //найти брата слева
     NodeTreeList *s=node->down;
     while(s!=NULL)
      if(s->right==son) break; //нашли левого брата
      else s=s->right;
     if(s==NULL) return; //Ошибка:  son не является сыном node
     else s->right=son->right;
   }
   DeleteSubTreeList(son->down,DeleteData); //удаление поддрева
   DeleteData(son->data);
   free(son);
  }
}
```

11. Функция левостороннего обхода для всех узлов поддерева node. При этом для каждого узла вызывается функция, адрес которой записан в указателе func.

```
void ForEachTreeListLeftNode(NodeTreeList *node,void(*func)(void*)){
  if(node==NULL) return;
  func(node->data);
  ForEachTreeListLeftNode(node->down,func);
  ForEachTreeListLeftNode(node->right,func);
}
```

12. Функция левостороннего обхода для всех узлов поддерева node. При этом для каждого листа вызывается функция, адрес которой записан в указателе func.

```
void ForEachTreeListLeftLeaf(NodeTreeList *node,void(*func)(void*)){
  if(node==NULL) return;
  if(node->down!=NULL)
    ForEachTreeListLeftLeaf(node->down,func);
 else
  func(node->data); //это лист
  if(node->right!=NULL)
    ForEachTreeListLeftLeaf(node->right,func);
}
```

13. Функция поиска на всех узлах дерева. На входе подается адрес поддерева, указатель на функцию сравнения. Параметр param. Функция Compare(node->data,param) на входе принимает адрес узла и указатель param, производит сравнение и если возвращает 1, то поиск прекращается. Функция Compare пишется в зависимости от целей поиска и структуры данных.

```
void* FindTreeListLeftNode(NodeTreeList *node, int (*Compare)
(void*,void*),void* param){
  if(node==NULL) return NULL;
  if(Compare(node->data,param)) return node->data;
  if(!FindTreeListLeftNode(node->down,Compare,param)) return node->data;
  return FindTreeListLeftNode(node->right,Compare,param);
}
```

14. Функция аналогична FindTreeListLeftNode только поиск осуществляется для листьев.

```
void* FindTreeListLeftLeaf(NodeTreeList *node,int(*Compare)
(void*,void*),void* param){
  if(node==NULL) return NULL;
  if(node->down!=NULL)
    FindTreeListLeftLeaf(node->down,Compare,param);
 else
  if(Compare(node->data,param) return node->data; //это лист
  if(node->right!=NULL)
    FindTreeListLeftLeaf(node->right,Compare,param);
}
```

7.2.2 Механизм конструирования деревьев на основе строки формата

Пусть имеется следующая грамматика, описывающая строку формата, задающую описание дерева в скобочной записи.

1. ***format=>node(node ,node,...node)***
2. node => node(node, node,...node)
3. node=>%n
4. node=>%v
5. node=>string

node может описывать быть строкой (string), параметр типа переменная string (%v), типа узел (%n).

Примеры записи формата

1. A(B,C(D,E)) — описание дерева без параметров.
2. D(X(%v,C)) — описывается дерево с одним параметром типа string.
3. V(X,%n,D(F,%v) — описывается дерево с двумя параметрами.

Функция преобразования строки формата в древовидный список

```
void TranslateAscizToTreeList(TranslateStateTreeList *ts,char *format,...);
```

1.TranslateAscizToTreeList(ts,"A(B,C(D,E))");

2.TranslateAscizToTreeList(ts, "D(X(%v,C))", "Hello");
3.TranslateAscizToTreeList(ts, "V(X,%n,D(F,%v)",node_ptr, "Hi");

Реализация

Размер вектора параметров

```
#define SIZEPARAMS 10
```

Объединение указателей

```
typedef union {
 void *pdata;                 //  v - указатель на любые данные
 NodeTreeList *pnode;    //  n - указатель на узел
} KruVariantTreeList;
```

Описание структуры состояния трансляции

```
typedef struct{
 void* (*CreateData)(char*);                          //функция работающая с
именем
 KruVariantTreeList params[SIZEPARAMS];   //вектор параметры
 int   size_params;                                   //количество параметров в
стеке
 char *string;                                        //описание дерева
 int   size_name;                                     //длина имени
 char *name;                                          //имя
 int   top_param;                                      //текущий параметр
 int   top;                                            //текущий символ в имени
 char  open_parenthes;                                //скобка (
 char  close_parenthes;                               //скобка )
 char  comma;                                          //запятая
 char  procent;                                        //заменитель процента
} TranslateStateTreeList;
```

Функция создание структуры TranslateStateTreeList

```
void InitTranslateStateTreeList(TranslateStateTreeList *ts,int
size_name,void*(CreateData(char*)))
{
 ts->CreateData=CreateData;
 ts->name=new char[size_name+1];
 ts->size_name=size_name;
 ts->open_parenthes='(';
 ts->close_parenthes=')';
 ts->comma=',';
 ts->procent='%';
}
```

Функция удаления структуры TranslateStateTreeList

```
void DeleteTranslateStateTreeList(TranslateStateTreeList *ts){
 delete []ts->name;
}
```

Функция переопределение разделителей

```
void SetPunctuators(TranslateStateTreeList *ts,char openp,char closep,char
comma,char procent){
 ts->open_parenthes=openp;  //переопределение скобки открывающей
 ts->close_parenthes=closep; //переопределение скобки закрывающей
 ts->comma=comma;
 ts->procent=procent;
}
```

Функция сообщения об ошибке

```
void TranslateError(char *err){
 printf("%s\n",err);
}
```

Функция конструирования древовидного списка

```
NodeTreeList *TranslateAscizToTreeList(TranslateStateTreeList *ts,char *for-
mat,...){
 ts->string=format;   //начальные установки
 char *ch=ts->string;
 ts->size_params=0;
```

```
 ts->top_param=0;
 va_list ap;
 va_start(ap,format);
 while(*ch) {    //взять все параметры из стека и занести в вектор параметров
  if(*ch==ts->procent){ //это процент (взять параметр)
    if(ts->size_params<SIZEPARAMS){
    ch++;
    switch(*ch){ //взять параметры из стека
     case 'n':{  //указатель на узел
       ts->params[ts->size_params++].pnode = va_arg(ap,NodeTreeList*);
       break;
       }
     case 'v':{  //указатель на любые данные
       ts->params[ts->size_params++].pdata = va_arg(ap,void*);
       break;
       }
     } //endswitch
    }
   } //endifch
   else ch++;
 }//end while

 NodeTreeList *node=AscizToTreeList(ts); //рекурсивный разбор строки
  return node;
}
```

Функция рекурсивного разбора строки и конструирования древовидного списка

```
NodeTreeList *AscizToTreeList(TranslateStateTreeList *ts){
  NodeTreeList *node=NULL;
  ts->top=0;
  while(*ts->string){ //выделить имя узла
   if(*ts->string==ts->open_parenthes) break; //это скобка (
  else
   if(*ts->string==ts->comma) break; //это запятая
  else
   if(*ts->string==ts->close_parenthes) break;
  else
   if(ts->top<ts->size_name-1) ts->name[ts->top++]=*ts->string++;
  else ts->string++; //если имя длинное просто пропускаем
```

```
 }
 //////////
 if(ts->top>0) {  //имя узла выделено
   ts->name[ts->top]=0;
   if(ts->name[0]==ts->procent){ //это параметр
    switch(ts->name[1]){         //определить тип параметра
     case 'v':         //создать новый узел
      if(ts->top_param<ts->size_params)
       node=CreateTreeListNode((void*)(ts->params[ts->top_param++].pdata));
       break;
     case 'n': //узел должен быть корнем (вставить поддерево)
      if(ts->top_param<ts->size_params)
       node=ts->params[ts->top_param++].pnode;
      break;
    }
   }
   else{ //это имя
    if(ts->CreateData!=NULL) //если функция есть
     node=CreateTreeListNode(ts->CreateData(ts->name));
     }
 }
 else{
   TranslateError("No name of node");
   return NULL;
   }
 //узел создан
 if(*ts->string=='\0') return node;
 else
  if(*ts->string==ts->open_parenthes){ //скобка открывающая
   ts->string++;
   node->down=AscizToTreeList(ts);
   if(*ts->string==ts->close_parenthes) ts->string++;
   }
 if(*ts->string==ts->comma){ //это запятая
    ts->string++;
    node->right=AscizToTreeList(ts);
    }
  if(*ts->string==ts->close_parenthes)return node;
  return node;
}
```

7.3 Использование кода Дьюи в древовидных списках

В конце 19 века американец Мельвиль Дьюи предложил универсальную десятичную классификацию. В 1876 г. вышло первое, очень краткое, издание таблиц этой классификации. Предложенный код Дьюи удобно использовать для указания узлов дерева. Например, корень дерева всегда 1., сыновья корня слева направо — 1.1., 1.2., 1.3. и т.д. Сыновья узла 1.2. будут иметь соответственно коды: 1.2.1, 1.2.2., 1.2.3. и т.д. Количество чисел в коде показывает глубину дерева.

Ниже предлагается механизм работы с древовидными списками на основе кода Дьюи.

Запишем ряд определений

1. Размер кода Дьюи, предполагается что глубина дерева не превышает 100.

```
#define CODEDEWEYSIZE  100
```

2. Задаем новый тип CodeDewey

```
typedef int CodeDewey[CODEDEWEYSIZE];
```

Запишем Функцию преобразования строки символов в код Дьюи (CodeDewey).

```
int* AscizToCodeDewey(CodeDewey c,char *str){ //преобразование строки в код дьюи
  int i=0;
  int k=0;
  char num[6];    //строка для выделения числа
  c[0]=1; c[1]=0; //начальные значения кода
  while(1){
   if(isdigit(*str)){
     if(i<5) num[i++]=*str; //здесь предполагается что число не превышает 5 рязрядов
     str++;
     }
  else
```

```
   if(*str=='.'){ //преобразовать в число
     num[i]=0; i=0;
     if(k<CODEDEWEYSIZE-1) c[k++]=atoi(num); //вызвать библиотечную
функцию atoi
      str++;
    }
  else
   if(*str==0){ //конец строки
      c[k]=0;  //признак конца
      break;
    }
  else str++;   //пропускаем
  }
  return &c[0];
}
```

Функция преобразования кода Дьюи в строку символов

```
void CodeDeweyToAsciz(char *str,CodeDewey c){ //обратное преобразование
 char snum[6];
 str[0]=0;
 for(int i=0; i<CODEDEWEYSIZE; i++){
  if(c[i]>0) {
   sprintf(snum,"%d.",c[i]);
   strcat(str,snum);
   }
  else break;
  }
}
```

Функция левостороннего обхода дерева с формированием кода Дьюи для каждого узла

Входные параметры:

NodeTreeList node — адрес корня древовидного списка;

void(*func)(void*,CodeDewey) — указатель на функцию обработки узла с заданным кодом Дьюи (ниже описан пример такой функции OutWith-Code);

CodeDewey c — код Дьюи узла;

int Level — переменная содержащая глубину рассматриваемого узла.

```
void ForEachTreeListLeftNodeWithCode(NodeTreeList *node,void(*func)
(void*,CodeDewey),CodeDewey c,int level){
  if(node==NULL) return;
  func(node->data,c);
  c[level+1]=1; c[level+2]=0;
  ForEachTreeListLeftNodeWithCode(node->down,func,c,level+1);
  c[level+1]=0; c[level]++;
  ForEachTreeListLeftNodeWithCode(node->right,func,c,level);
}
```

Функция поиска узла с заданным кодом Дьюи

```
NodeTreeList* FindTreeListNode(KruTreeList *tree,CodeDewey c){
  int i=0;
  int cont=c[i];
  NodeTreeList*addr=tree->root;
  while(i<CODEDEWEYSIZE){
   for(int k=0; k<(c[i]-1); k++){ //движение вправо
    if(addr==NULL) return NULL;
     if(addr->right!=NULL) addr=addr->right;
    else return NULL;
   }
   i++;
   if(c[i]==0) return addr; else addr=addr->down;
   }
  return addr;
}
```

Функция для запуска алгоритма ForEachTreeListLeftNodeWithCode

```
void OutWithCode(void *str,CodeDewey c){
 char sCode[100];
 CodeDeweyToAsciz(sCode,c);
 printf("%s %s\n",sCode,(char*)str);
}
```

Тестовый пример показывающий работу описанных функций

```
char *classif[] = { "Enimal",
                    "xxxx",
                    "yyyy",
                    "zzzzz",
```

```
                "kkkkk"
                 };
void KruTestTreeListDewey(){
  char sCodeDewey[100];
  KruTreeList tree;
  CodeDewey code;
  AscizToCodeDewey(code,"1.333.5.6.80.23");
  CodeDeweyToAsciz(sCodeDewey,code);
  printf("%s\n",sCodeDewey);
  tree.root=CreateTreeListNode((void*)classif[0]);
  AscizToCodeDewey(code,"1.");
  NodeTreeList* addr=FindTreeListNode(&tree,code);
  printf("Add -> %s\n",(char*)(addr->data));
  for(int i=1; i<3; i++)
    AddTreeListNodeRightSon(addr,(void*)classif[i]);
  AscizToCodeDewey(code,"1.1.");
  addr=FindTreeListNode(&tree,code);
  printf("Add-> %s\n",(char*)(addr->data));
  for(int i=3; i<5; i++)
    AddTreeListNodeRightSon(addr,(void*)classif[i]);
  ForEachTreeListLeftNode(tree.root,Out);
  printf("____________________\n");
  ForEachTreeListLeftLeaf(tree.root,Out);
  AscizToCodeDewey(code,"1.");
  ForEachTreeListLeftNodeWithCode(tree.root,OutWithCode,code,0);
  printf("____________________\n");
  while(!kbhit());

}


void Out(void* str){
 printf("%s\n",(char*)str);
}
```

8 ДЕРЕВЬЯ И/ИЛИ

8.1 Основные понятия

Деревом И/ИЛИ является дерево, которое имеет два типа узлов (И-узел и ИЛИ-узел). Впервые понятие И/ИЛИ дерева появилось в работе Слайгла [ ]. ИЛИ-узел означает, что задача может быть решена, если решать задачу 1, или задачу 2, или задачу 3 (рис. 8.1).

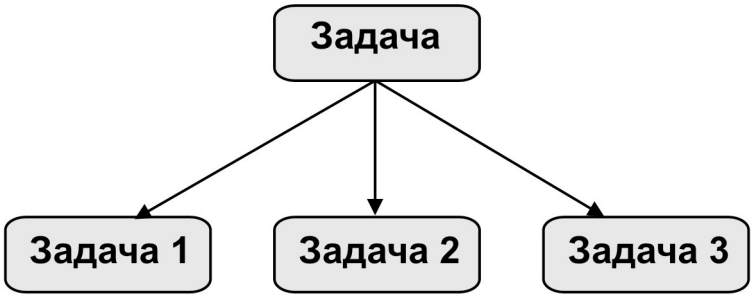


Рис. 8.1 — Узел ИЛИ

И-узел означает, что задачу можно разложить на подзадачи и, решив все задачи из этого списка, получить решение исходной задачи (рис. 8.2).

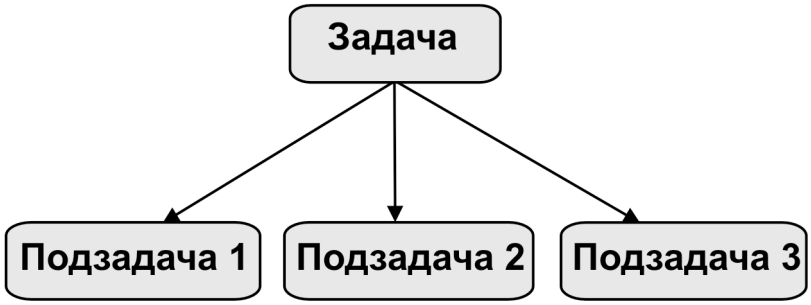


Рис. 8.2 — Узел И

Вариантом дерева И/ИЛИ назовем поддерево, которое получается из заданного путем отсечения выходных дуг кроме одной, у всех ИЛИ-узлов. Вариант в терминах решения задачи задает вариант решения задачи.

Деревья И/ИЛИ получили распространение в исследованиях по искусственному интеллекту [44–47]. Ниже предлагаются эффективные алгоритмы для перечисления вариантов в дереве И/ИЛИ [14].

8.2 Алгоритм подсчета вариантов

Рассмотрим алгоритм подсчета вариантов решений в дереве И/ИЛИ. Для этого запишем следующую рекурсивную функцию:

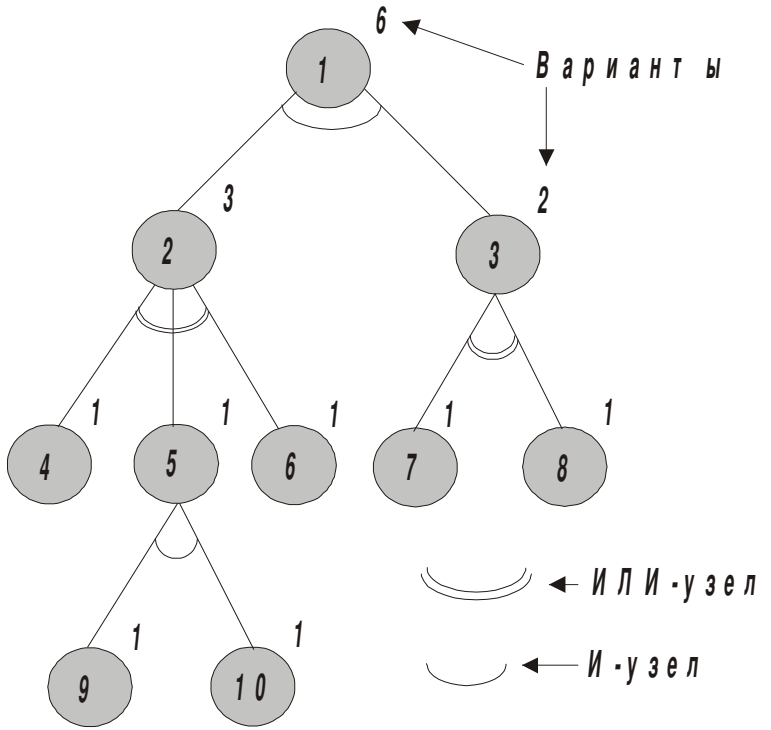
$$\varpi(z) = \begin{cases} \prod_{i=1}^{n} \varpi(s_i^z), & \text{для И-узла;} \\ \sum_{i=1}^{n} \varpi(s_i^z), & \text{для ИЛИ-узла;} \\ 1, & \text{для листа,} \end{cases} \qquad (2.10)$$

где z — рассматриваемый узел дерева;

$\{ s_i^z \}$ — множество сыновей узла z;

n — количество сыновей.

Подсчитав значение функции для корня дерева, можно получить общее число вариантов подстановок, имеющихся в данном дереве. При этом



будет подсчитано количество вариантов подстановок для каждого узла всего дерева. Пример подсчета вариантов показан на рис. 8.3.

Рис. 8.3 — Пример подсчета вариантов в И/ИЛИ дереве

8.3 Алгоритм нумерации вариантов

Далее, используя значения функции для каждого узла, можно построить алгоритм нумерации вариантов. Этот алгоритм по номеру из данного дерева получает необходимый вариант подстановки. Предварительно зададим две функции нумерации, которые по номеру варианта для рассматриваемого узла вычисляют номера вариантов для сыновей. Для И-узла будет следующая функция:

$$L_i^s(L^z) = \begin{cases} (L^z / \prod_{j=1}^{i} \varpi(s_j^z)) \bmod \varpi(s_i^z), & i > 1; \\ L^z \bmod \varpi(s_i^z), & i = 1, \end{cases} \qquad (2.11)$$

где z — рассматриваемый узел дерева;

$\{ s_i^z \}$ — множество сыновей узла z;

$\omega(S_i^Z)$ — количество вариантов в поддереве с узлом S_i^Z.

Примечание. При выполнении операции деления используются алгоритмы целочисленной арифметики.

Рассмотрим пример использования этой функции. Пусть задан И-узел z, у которого имеется три сына (S_1, S_2, S_3), и известно количество вариантов $\omega(z) = 6$, $\omega(S_1) = 1$, $\omega(S_2) = 3$, $\omega(S_3) = 2$.

Тогда можно записать следующую таблицу значений номеров вариантов для сыновей в зависимости от номера варианта узла z:

L_z	$L(S_1)$	$L(S_2)$	$L(S_3)$
0	0	0	0
1	0	1	0
2	0	2	0
3	0	0	1
4	0	1	1
5	0	2	1

Для ИЛИ-узла необходимо определить не только номер варианта, но и номер соответствующего сына. Для этого запишем следующую функцию:

$$L^s_{k+1}(L^z) = \begin{cases} L^z \quad \text{ï ðè} \quad L^z < \varpi(s_1^z), \quad k = 0; \\ \min\limits_k (L^z - \sum\limits_{j=1}^{k} \varpi(s_j^z)) \quad \text{ï ðè} \quad L^s_{k=1}(L^z) \sigma 0, \end{cases} \qquad (2.12)$$

где z — рассматриваемый узел дерева;

$\{ s_i^z \}$ — множество сыновей узла z;

$\omega(S_i^Z)$ — количество вариантов в поддереве с узлом S_i^Z.

Рассмотрим пример. Пусть задан ИЛИ-узел с 3 сыновьями и $\omega(z) = 9$, $\omega(S_1) = 3$, $\omega(S_2) = 2$, $\omega(S_3) = 4$, $\omega(z) = \omega(S_1) + \omega(S_2) + \omega(S_3)$.

Тогда можно записать следующую таблицу значений для номера сына и номера варианта в зависимости от значения варианта узла z:

L_z	K	L_{k+1}	Сын
0	0	0	S_1
1	0	1	S_1
2	0	2	S_1
3	1	0	S_2
4	1	1	S_2
5	2	0	S_3
6	2	1	S_3
7	2	2	S_3
8	2	3	S_3

Алгоритм определения варианта подстановки по заданному номеру будет следующий.

1. Первоначально производится подсчет количества вариантов для каждого узла дерева.

2. В вариант записывается корень дерева, и он становится текущим узлом.

3. Определяется тип текущего узла. Если это И-узел, то переход на шаг 4, иначе переход на шаг 5.

4. Все сыновья рассматриваемого узла записываются в данный вариант подстановки и для них соответственно с функцией для И-узлов определяются номера вариантов для соответствующих поддеревьев (см. 2.11).

5. Если это ИЛИ-узел, то определяется единственный сын и номер варианта для соответствующего поддерева (см. 2.12 для определения сына и номера варианта для ИЛИ-узла).

6. Происходит переход на рассмотрение следующего узла из списка включенных в данный вариант подстановок.

7. Процесс определения варианта подстановок будет происходить до тех пор, пока не будут достигнуты листья для всех рассматриваемых узлов варианта подстановок.

На рис. 8.4 показаны все варианты для дерева И/ИЛИ, приведенного в разделе 2.7.4.

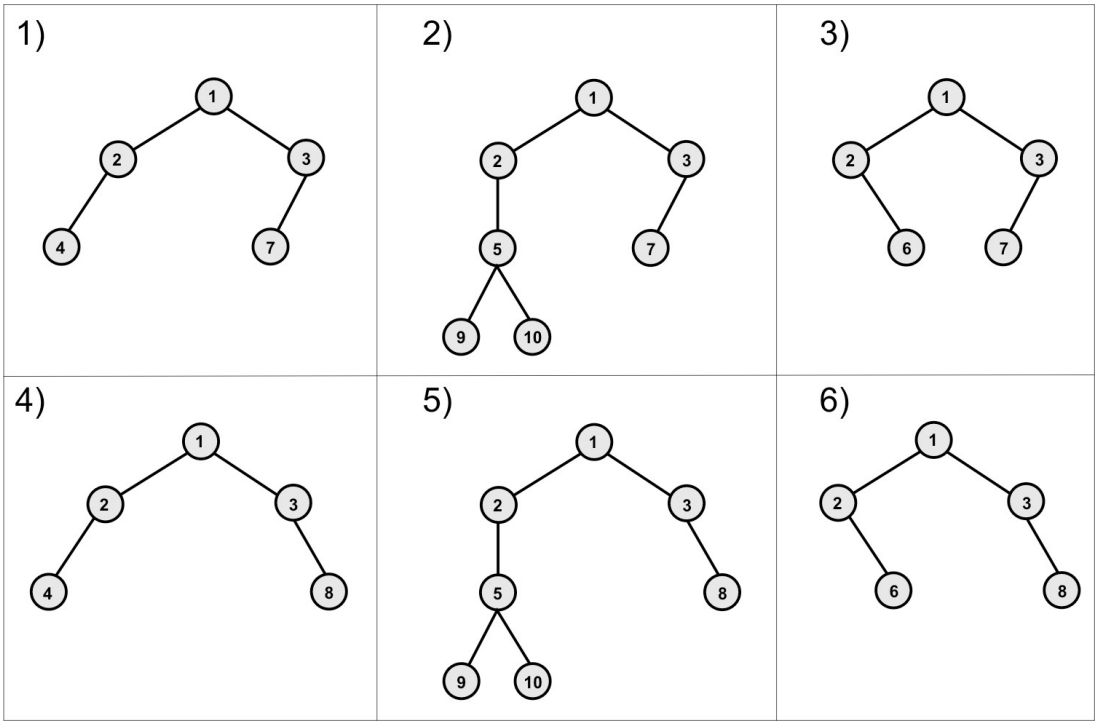


Рис. 8.4 — Все варианты для примера И/ИЛИ дерева

8.4 Программная реализация

В основе программной реализации лежит контейнер древовидного списка (см. раздел «Организация древовидного списка»). Перечислим константы:

Тип узла И

```
#define AND_NODE 1
```

Типа узла ИЛИ

```
#define OR_NODE  2
```

Описание структуры узда дерева И-ИЛИ

```
typedef struct{    //описание узла
  int type;          //тип узла
  int var;           //количество вариантов
  void* data;        //данные
} KruAOTreeNode;
```

Описание структуру дерева И-ИЛИ

```
typedef struct{       //заголовок дерева
  NodeTreeList *root; //корень
  int size;           //кол-во узлов
  int var;            //кол-во вариантов
} KruAOTree;
```

Функция инициализация структуры дерева И-ИЛИ

```
void InitKruAOTree(KruAOTree *aot){
 aot->root=NULL;
 aot->size=0;
 aot->var=0;
}
```

Функция создание узла дерева

```
KruAOTreeNode* CreateKruAOTreeNode(int type,void* data){
  KruAOTreeNode *n=new KruAOTreeNode;
  if(n==NULL) return NULL;
  n->var=0;
  n->data=data;
  n->type=type;
  return n;
}
```

Функция вывода узла дерева И-ИЛИ в случае, когда по указателю data хранится строка символов

```
void OutputKruAOTreeNode(void *n){
 printf("%d %d %s\n",((KruAOTreeNode*)n)->type,((KruAOTreeNode*)n)-
>var,(char*)((KruAOTreeNode*)n)->data);
 //printf("%s\n",(char*)((KruAOTreeNode*)n)->data);
```

```
}
```

Функция удаления узла дерева И-ИЛИ в случае, когда по указателю data хранится строка символов

```
void DeleteKruAOTreeNode(void *n){
  delete [](KruAOTreeNode*)n)->data;
  delete (KruAOTreeNode*)n;
}
```

Функция создания узла дерева И-ИЛИ по описанию в name. Структура name должна быть следующей:

<Символ типа узла><строка>

<Символ типа узла> может быть

1) A — для И-узла;
2) O — для ИЛИ-узла.

Эта функция используется при трансляции скобочной записи дерева в древовидный список

```
void* CreateKruAOTreeNode(char *name){
   KruAOTreeNode *n=new KruAOTreeNode;
   if(n==NULL) return NULL;
   if(*name=='A')  n->type=AND_NODE;
    else n->type=OR_NODE;
   n->var=0;
   char *s=new char[strlen(&name[1])+1];
   strcpy(s,&name[1]);
   n->data=s;
   return n;
}
```

Функция рекурсивного подсчета вариантов в дереве И-ИЛИ

```
int CountVarAOTree(NodeTreeList *node){
   KruAOTreeNode *d;
   int res;
   if(node==NULL) return 0;
   d=(KruAOTreeNode*)node->data;
    if(node->down!=NULL){
    NodeTreeList *t=node->down;
```

```
  if(d->type==AND_NODE)  res=1; else res=0; //установить начальные значения для res
   while(t!=NULL){ //для всех сыновей
    if(d->type==AND_NODE)
        res=res*CountVarAOTree(t); //рекурсивный подсчет для узла И
    else res=res+CountVarAOTree(t); //рекурсивный подсчет для узла ИЛИ
    t=t->right;
   }
  }
 else res=1; //это лтст
 d->var=res; //установить значение var для данного узла
 return res;
}
```

Функция обхода узлов варианта с заданным номером, node — корень дерева, number номер варианта, func — указатель на функцию обработки узла. Функция работает по алгоритму перечисления вариантов дерева И-ИЛИ

```
void ForEachVarAOTree(NodeTreeList *node, int number, void (*func)(void*))
{
  KruAOTreeNode *d;
  int res;
  if(node==NULL) return;
  d=(KruAOTreeNode*)node->data;
  if(func!=NULL) func(d); //вызов функции
  if(node->down!=NULL){
   NodeTreeList *t=node->down;
   if(d->type==AND_NODE)  res=1; else res=0;
   while(t!=NULL){
    KruAOTreeNode *td=(KruAOTreeNode*)t->data;
    if(d->type==AND_NODE){ //для всех И-узлов
     ForEachVarAOTree(t,number%td->var,func);
     number/=td->var;
    }
    else {  //для одного ИЛИ-узла
     if(number<td->var) { ForEachVarAOTree(t,number,func); break;}
     number-=td->var;
    }
    t=t->right;
   }
  }
```

```
}
```

Функция обхода узлов и обработки листьев варианта с заданным номером, node — корень дерева, number номер варианта, func — указатель на функцию обработки листа. Функция работает по алгоритму перечисления вариантов дерева И-ИЛИ.

```
void ForEachVarAOTreeLeaf(NodeTreeList *node, int number, void (*func)
(void*)){
  KruAOTreeNode *d;
  int res;
  if(node==NULL) return;
  d=(KruAOTreeNode*)node->data;
  if(node->down==NULL){ //это лист
   if(func!=NULL) func(d); //вызов функции
   }
  else {
  NodeTreeList *t=node->down;
  if(d->type==AND_NODE)  res=1; else res=0;
  while(t!=NULL){
   KruAOTreeNode *td=(KruAOTreeNode*)t->data;
    if(d->type==AND_NODE){ //для всех И-узлов
     ForEachVarAOTree(t,number%td->var,func);
     number/=td->var;
    }
   else {  //для одного ИЛИ-узла
    if(number<td->var) { ForEachVarAOTreeLeaf(t,number,func); break;}
    number-=td->var;
   }
   t=t->right;
  }
 }
}
```

Тестовая функция для показа организации и работы с деревьями И-ИЛИ

```
void TestKruAOTree(){
// конструирование дерева
 TranslateStateTreeList tsv; //создаем структуру
 InitTranslateStateTreeList(&tsv,20,CreateKruAOTreeNode);  //инициализиру-
ем
```

```
//трансляция  описание дерева И_ИЛИ (см. рис)
// в древовидный список
NodeTreeList *ao_tree=TranslateAscizToTreeList(&tsv,
 "A1._node-1\
(O1.1_node-2(\
A1.1.1_node-4,\
O1.1.2._node_5(\
A1.1.2.1_node-9,\
O1.1.2.2._node_10),\
A1.1.3_node-6),\
O1.2_node-3(\
A1.2.1_node-7,\
A1.2.2_node-8)");
//подсчет количества вариантов
 int nvar=CountVarAOTree(ao_tree);
 printf("++++ var =%d\n",nvar);
//левосторонний обход дерева И-ИЛИ печать узлов
 ForEachTreeListLeftNode(ao_tree,OutputKruAOTreeNode);
 printf("++++++++++++++++++\n");
//левосторонний обход дерева И-ИЛИ печать листьев
 ForEachTreeListLeftLeaf(ao_tree,OutputKruAOTreeNode);
 //перечисление вариантов в дереве и печать
 for(int num=0; num<nvar; num++){
   ForEachVarAOTree(ao_tree,num,OutputKruAOTreeNode);
   printf("_________%d\n",num);
 }

//очищение динамической памяти
 DeleteTranslateStateTreeList(&tsv);
 DeleteSubTreeList(ao_tree,DeleteKruAOTreeNode);

 while(!kbhit());
}
```

Листинг работы тестовой программы

```
++++ var =8
1 8 1._node-1
2 4 1.1_node-2
1 1 1.1.1_node-4
2 2 1.1.2._node_5
1 1 1.1.2.1_node-9
```

2 1 1.1.2.2._node_10
1 1 1.1.3_node-6
2 2 1.2_node-3
1 1 1.2.1_node-7
1 1 1.2.2_node-8
++++++++++++++++
1 1 1.1.1_node-4
1 1 1.1.2.1_node-9
2 1 1.1.2.2._node_10
1 1 1.1.3_node-6
1 1 1.2.1_node-7
1 1 1.2.2_node-8
1 8 1._node-1
2 4 1.1_node-2
1 1 1.1.1_node-4
2 2 1.2_node-3
1 1 1.2.1_node-7
_____0
1 8 1._node-1
2 4 1.1_node-2
2 2 1.1.2._node_5
1 1 1.1.2.1_node-9
2 2 1.2_node-3
1 1 1.2.1_node-7
_____1
1 8 1._node-1
2 4 1.1_node-2
2 2 1.1.2._node_5
2 1 1.1.2.2._node_10
2 2 1.2_node-3
1 1 1.2.1_node-7
_____2
1 8 1._node-1
2 4 1.1_node-2
1 1 1.1.3_node-6
2 2 1.2_node-3
1 1 1.2.1_node-7
_____3
1 8 1._node-1
2 4 1.1_node-2
1 1 1.1.1_node-4

2 2 1.2_node-3
1 1 1.2.2_node-8
_____4
1 8 1._node-1
2 4 1.1_node-2
2 2 1.1.2._node_5
1 1 1.1.2.1_node-9
2 2 1.2_node-3
1 1 1.2.2_node-8
_____5
1 8 1._node-1
2 4 1.1_node-2
2 2 1.1.2._node_5
2 1 1.1.2.2._node_10
2 2 1.2_node-3
1 1 1.2.2_node-8
_____6
1 8 1._node-1
2 4 1.1_node-2
1 1 1.1.3_node-6
2 2 1.2_node-3
1 1 1.2.2_node-8
_____7

9 ТРАНСЛЯТОРЫ, ИНТЕРПРЕТАТОРЫ, АССЕМБЛЕРЫ, ВИРТУАЛЬНЫЕ МАШИНЫ, ДИЗАССЕМБЛЕРЫ И ОТЛАДЧИКИ

9.1 Основные определения

При создании сложных программных систем зачастую приходится писать модули, которые или транслируют или интерпретируют выражения некоторого языка. Кроме того, знание принципов построения и работы таких программ позволяет лучше использовать имеющиеся компиляторы и другие инструментальные средства.

Для построения и анализа трансляторов разработана теория, методы, алгоритмы, инструментальные средства генерации компиляторов и анализа их функционирования. Все это можно найти в литературе приведенной в конце раздела. В этой главе будут на конкретных достаточно простых примерах показаны идеи, принципы, методы и технологии разработки трансляторов, компиляторов, интерпретаторов, виртуальных машин, ассемблеров, линкеров, дизассемблеров и отладчиков.

Дадим ряд определений:

Транслятор — программа преобразования описания с одного языка на другой.

Компилятор программа перевода с алгоритмического языка высокого уровня на язык машинных команд.

Ассемблер — программа перевода с низкоуровневого языка описания в последовательность машинных команд.

Интерпретатор — программа исполнения программы написанной на алгоритмическом языке.

Виртуальная машина — программа, моделирующая работу некоторой вычислительной машины.

9.2 Метод рекурсивного спуска

Язык арифметических выражений достаточно прост и практически всем понятно, как записывать эти выражения. Арифметические выражения строятся из идентификаторов, констант, знаков операций и скобок, кроме того можно записывать еще и некоторый функции, например, корень квадратный, sin, cos и т.д.

Примеры арифметических выражений:

1. x+1/(10+y)
2. c+sin(a*x+2)
3. (count+1)*(count-1)

Грамматика языка арифметических выражений записана ниже.

<E> => <T>+<E>
<E> => <T>-<E>
<E> => <T>
————————
<T> => <D>*<E>
<T> => <D>/<E>
<T> => <D>
————————
<D> => (<E>)
<D> => <идентификатор>
<D> => <константа>

Идентификатор — последовательность букв и цифр, начинающаяся с буквы.

Константа — вещественное число.

Не вдаваясь в теоретические подробности, по данной грамматике можно построить алгоритм разбора, который основан на методе рекурсивного спуска[]. Суть этого метода заключается в следующем: пишутся три функции, для групп выделенных правил.

```
01  Функция РазборE()
02    вызвать РазборT()
03    цикл
04      если на входе "+", то сдвиг и вызвать  РазборT()
05     иначе
06      если на входе "-", то сдвиг и вызвать  РазборT()
07     иначе выход из цикла
08    конец цикла
09  конец функции РазборE
10
11  Функция РазборT()
12      вызвать РазборD()
13      цикл
14       если на входе "*" , то сдвиг и вызвать  РазборD()
15      иначе
16       если на входе "/", то сдвиг и вызвать  РазборD()
17      иначе выход из цикла
18      конец цикла
19  конец функции РазборT
20
21  Функция РазборD()
22    если на входе "(" , то сдвиг и вызвать  РазборE(), ждать " )"
23   иначе
24    если на входе буква, то взять идентификатор
25   иначе
26   если на входе цифра, то взять константу
27  иначе выдать ошибку, сдвиг входа
28  конец функции РазборD
```

Рассмотрим работу метода рекурсивного спуска на примере.

Пусть есть входная строка

x+10*(a+1)

Первоначально начинает работу функция ***РазборE***, она вызывает функцию ***РазборT*** (строка 2). Функция ***РазборT*** вызывает ***РазборD*** (строка 12). Функция ***РазборD*** проверяет на входе открывающую скобку, на входе символ ***"x"*** — это буква, значит следует идентификатор (см. строка 24). Функция ***РазборD*** берет идентификатор ***x*** и сдвигает вход на следующий символ. После вызова функции ***РазборD*** управление возвращается на строку 22 функции ***РазборT.*** На входе при этом стоит символ ***"+",*** поэтому происходит выход из цикла и возврат в функцию ***РазборE*** на выполнение строки 3. Входим в цикл, проверяем на входе ***"+",*** да — сдвигаем вход на следующий символ (*"1"*) и вызываем снова функцию ***РазборT.*** Та в свою, очередь вызывает ***РазборD***, которая забирает из входной строки константу *"10"*, сдвигает вход (на входе будет символ ***"*"***) и возвращает управление в функцию ***РазборT*** в строку 13, где записан цикл, входим в цикл и проверяем на знак умножить. Да совпало, сдвигаем вход (на входе будет символ ***")"***) и вызываем функцию ***РазборD***. Проверяем на входе стоит скобка открывающая — да совпало, сдвигаем вход и вызываем функцию ***РазборE*** (вот она рекурсия, функция ***РазборE*** вызывается второй раз) . Функция ***РазборE*** совместно с функциями ***РазборT*** и ***РазборD*** производит разбор выражения в скобках, после завершения последнего вызова функции ***РазборE*** на входе будет закрывающая скобка и управление будет передано в строку 22, сразу после вызова ***РазборE,*** где ожидается закрывающая скобка. Далее производится сдвиг и возврат в функцию ***РазборT***, откуда возврат в функцию ***РазборE*** в строку 04, поскольку на входе больше ничего нет функция ***РазборE*** завершает работу.

9.3 Реализация интерпретатора арифметических выражений

9.3.1 Организация хранения идентификаторов, их значений и констант

Для хранения идентификаторов, их значений и констант предлагается использовать таблицу, представленную как массив структур, описание которого представлено ниже.

```
#define SIZETABLVAR  30            //размер таблицы символов
struct {                           //описание таблицы символов
  int length;                      //длина имени
  char Name[10];                   //имя переменной
  double init;                     //значение переменной
      } TablVar[SIZETABLVAR];
int TopVar;                        //номер свободной ячейки таблицы
```

Это массив структур статический. Поэтому вся память при запуске программы обнуляется.

Кроме того, имеется две функции FindNameVar и AddNameVar. Первая ищет идентификатор таблице и если находит, то возвращает номер элемента таблицы хранящий этот идентификатор, соответственно и его значение. Вторая — производит добавление нового идентификатора в таблицу и возвращает его табличный номер. Если идентификатор хранится уже в таблице, то возвращает номер идентификатора.

```
int FindNameVar(char *var) { //найти имя в таблице символов
  int i;
  int len;
  len=strlen(var);
  for(i=0; i<SIZETABLVAR; i++) { //просмотреть всю таблицу
      if(TablVar[i].length==0) return -1; //не найдено
      if(TablVar[i].length==len)
      if(strcmp(TablVar[i].Name,var)==0) return i; //найдено
   }
   return -1; //не найдено
}

int AddNameVar(char *var) { // добавить новое имя в таблицу символов
  int num;
  int len;
  char *ptr;
  if((num=FindNameVar(var))==-1)  { //такое имя не найдено
    if(TopVar<SIZETABLVAR)   { //вставка имени
      num=TopVar;                    //взять номер свободного элемента массива
      len=strlen(var);               //вычислить размер идентификатора
      TablVar[TopVar].length=len;
      strncpy(TablVar[TopVar].Name,var,9);
      TopVar++;
```

```
    }
    else Error("Переполнена таблица символов");
  }
 return num; //вернуть индекс имени в таблице
}
```

Хранение констант в таблице символов организовано следующим образом. Выделенная строка символов, в которой записана константа, хранится как имя идентификатора, а ее значение записывается как значение идентификатора. Например, TablVar[i].Name=”3.355” это будет имя. А TablVar[i].init=3.335 — хранит значение.

9.3.2 Использование стека для выполнения арифметических операций

Стек является удобным инструментом для выполнения арифметических операций.

Например, для выполнения операции сложить необходимо знать адрес первого слагаемого, адрес второго слагаемого и адрес, куда записать результат. При организации стековых вычислений значения операндов берутся из стека и результат операции также заносится в стек. Например, C=A+B, тогда стековые операции будут выглядеть так:

Положить(A), Положить(B), Сложить, Взять(C)

Или для формулы (A+B)*(X-Y)+20

Положить(A), Положить(B), Сложить

Положить(X), Положить(Y), Вычесть

Умножить

Положить (20), Сложить

Результат вычисления этого выражения будет храниться на вершине стека.

Ниже описан простой вариант организации стека и стековых операций положить и взять.

Описание статической памяти

```
#define SIZESTACK       30          //размер стека
double Stack[SIZESTACK];            //стек (массив фиксированного размера)
int TopStack;                       //указатель вершины стека
```

Переменная TopStack всегда указывает на первый свободный элемент стека, а последний занесенный элемент хранится по адресу TopStack-1. Стек пустой если переменная TopStack равна нулю. Стек переполнен если TopStack равна SIZESTACK. Ниже приведен текст функций Push и Pull — первая кладет значение в стек, второй выталкивает значение из стека.

```
//
void Push(double f) { // положить значение в стек
 if(TopStack<SIZESTACK)
  Stack[TopStack++]=f;
  else Error("Стек переполнен");
}
//
double Pull() {          //взять значения из стека
 if(TopStack>0) {
   TopStack--;
   return Stack[TopStack];
  }
 return 0;
}
```

9.3.3 Вспомогательные функции

К вспомогательным функциям относятся функции Error, SkipBlanks, GetName, getfloat, match. Необходимо отметить, что почти все указанные функции работают с глобальным указателем на текущий символ входной строки. Он объявлен и проинициализированным следующим образом:

```
char *ch; //объявление глобального указателя
ch=in_line[0]; .//инициализация
```

Функция Error обеспечивает вывод сообщения на экран терминала. Ниже приведен текст этой функции:

```
void Error(char *soo){ //вывод сообщения об ошибке
 printf("%s\n",soo);
}
```

Функция SkipBlanks обеспечивает пропуск служебных символов во входной строке. Пропуск обеспечивает тем, что статический указатель ***ch*** на текущий символ увеличивается на еденицу. Это процесс продолжается

пока текущий символ будет служебным (табуляция, конец файла, возврат каретки, перевод строки, пробел). Ниже приведет текст функции

```
void SkipBlanks() { // пропуск  служебных символов
 while(1) { //цикл пока есть служебное символ
  switch (*ch)  {
   case 8:      //табуляция
   case 26:     //конец файла
   case 10:     //возврат каретки
   case 13:     //перевод строки
   case ' ':       //пробел
             ch++; break;
   default: //если это не служебный символ или конец строки
     return;
  }
 }
}
```

Функция GetName обеспечивает выделение идентификатора из входной строки.

Первоначально функция вызывает SkipBlanks для пропуска служебных символов. Затем с помощью функции isalpha описанной в заголовочном файле ctype.h проверяется, является ли текущий символ буквой (буковой являются буквы латиницы). Далее в цикле выбираются буквы или цифры (для проверки используется библиотечная функция isalnum, описанная в ctype.h)

```
int GetName(char *name, int max) { // взять имя во входной строке
  int i;
  SkipBlanks();           //пропустить пробелы
  if(isalpha(*ch)) *name++=*ch++;   //первый символ должен быть буквой
  else { *name=0; return 0;}
  for(i=1; i<max; i++) {    //взять остальные символы (буква или цифра)
     if(isalnum(*ch))  *name++=*ch++;
   else {
     *name=0;
      return 1;
    }
  }
  *name=0; Error("Большое имя"); return 0;
}
```

Функция getfloat выделяет из входной строки константу, записывает в строку num и преобразует ее в вещественное число с помощью библиотечной функции atof, которая описана в заголовочном файле stdlib.h

```
double getfloat(char *num, int max) { // взять число во входной строки и
                                      // преобразовать к double
 int i=0, j;
 SkipBlanks();             //пропустить пробелы
 if(*ch=='-'|| *ch=='+') //взять знак
              num[i++]=*ch++;
 SkipBlanks();             //пропустить пробелы
 for( ; i<max; i++) {     //взять целую часть числа
   if(isdigit(*ch)) num[i]=*ch++;
  else break;
 }
 j=i;
 if(*ch=='.') {           //взять дробную часть числа
    num[i]=*ch++;
   for(j=i+1; j<max; j++) {
    if(isdigit(*ch)) num[j]=*ch++;
     else break;
   }
  }
 num[j]=0;
 return atof(num); //это стандартная функция Си описана в <stdlib.h>
}
```

Функция match производит сопоставление строки str с символами входной строки, начиная с текущего (по указателю ch). Если строка str полностью содержится в во входной строке, то указатель ch сдвигается на заданной количество символов. В противном случае не сдвигается.

```
int match(char *str) { //эта функция разбиралась в разделе "Обработка
строк"
  char *chv;
  SkipBlanks(); //пропустить служебные символы
  chv=ch;          //запомнить указатель
  while(*str)  { //цикл проверки совпадения
   if((*str++)!=(*chv++)) return 0; //вернуть ноль если не совпала
  }
  ch=chv; //переместить указатель
```

```
 return 1; //вернуть значение совпадения
}
```

9.3.4 Реализация метода рекурсивного спуска

Рекурсивный разбор арифметического выражения начинается с работы функции ExpressAdd, которая осуществляется разбор и вычисление аддитивных составляющих арифметического выражения.

TranSin — функция разбора и вычисления синуса;

ExpressFunc — разбор первичного (см. алгоритм ***РазборD***());

ExpressMult — разбор и вычисление выражений с умножением и делением;

ExpressAdd — разбор и вычисление выражений со сложением и вычитанием;

AssignOp — вычисление и разбор выражения и присвоение результат переменной.

Текст функции разбора и вычисления синуса

```
void TranSin(void) { // вычисление синуса
 double f;
 if(match("(")) {  //есть ли скобка ?
   ExpressAdd();   //вычислить выражение аргумента синуса
   f=Pull();       //взять значение из стека
   Push(sin(f));   //положить в стек значения синуса
   match(")");     //ждать закрывающуюся скобку
  }
  else Error("Ошибка в функции sin");
}


void ExpressFunc(void) {// вычисление первичного выражения
  int num;
  char name[30];
  double f;
  int max=29;
  if(match("(")){  //вычисление выражения в скобках
     ExpressAdd(); //разобрать и вычисллить выражение в скобках
     if(match(")")==0) Error("Пропущена )");
   }
  else { //вычисление функций
```

```
     if(match("sin")) TranSin();
   else { //это переменная?
     if(GetName(name,max)==1)  { //взятие значения переменной
       num=AddNameVar(name);     //найти  переменную в таблице
       Push(TablVar[num].init);        //положить табличное значение в стек
      }
     else {  //взять значение константы
       f=getfloat(name,max);
       Push(f);                //положить значение константы в стек
     }
   }
  }
}

void ExpressMult(void) { //вычисление выражения для умножения и деления
 double f1, f2;
 ExpressFunc();                //вычислить первичное
 while(1)  {
  if(match("*")) {         //если знак умножения
    ExpressFunc();         //вычислить второе первичное
    f1=Pull(); f2=Pull(); //взять из стека множители
    Push(f1*f2);            //положить в стек произведение
   }
 else
  if(match("/")){    //если знак деления
    ExpressFunc();  //вычислить второе первичное
    f1=Pull();          //взять делитель
    f2=Pull();          //взять делимое
    if(f1==0.) Error("Деление на 0");
    else Push(f2/f1);  //положить частное в стек
   }
  else break;
  }
}

void ExpressAdd(void) { // вычисление выражения для сложения и  вычитания
 double f1, f2;
 ExpressMult();            //вычислить значение первого операнда
 while(1) {
  if(match("+")){          //если знак плюс
```

```
   ExpressMult();          //вычислить значение для второго операнда
   f1=Pull(); f2=Pull(); //взять значения слагаемых из стека
   Push(f1+f2);            //положить сумму в стек
  }
 else
  if(match("-")) {     //если это минус
   ExpressMult();    //взять значение второго операнда
   f1=Pull(); f2=Pull(); //взять значения из стека
   Push(f2-f1);          //положить разность в стек
  }
 else break;
 }
}

void AssignOp(void) {// выполнение операции присваивания
 char name[10];
 int max=9;
 int num;
 if(GetName(name,max)==1) {  //взять имя переменной
       num=AddNameVar(name); //записать имя в таблицу
    }
    else  {
      ExpressAdd();       //иначе вычислить выражение
      printf("%f\n",Pull());
      return;
     }
  if(match("=")) {         //имеется операция присваивания
     ExpressAdd();        //вычислить выражение
     TablVar[num].init=Pull(); //записать значение в таблицу
   }
  printf("%f\n",TablVar[num].init);
}

PrintTablVar() {// вывод таблицы символов со значениями
 int i;
 for(i=0; i<TopVar; i++)  {
  printf("%2d %10s   %f\n",i,TablVar[i].Name,TablVar[i].init);
  }
}

void main() {
```

```
 TopVar=0;          //инициализация стека и таблицы переменных
 TopStack=0;
 while(1) {
  printf("calc>");    //вывод подсказки
  gets(in_line);        //ввод строки символов
  ch=&in_line[0];     //инициализация указателя разбора
   if(match("end"))  //если это "end", то завершение работы
     break;
  else
  if(match("tabl"))  //если это "tabl", то вывести таблицу символов
     PrintTablVar();
  else                  //иначе вычислить значение выражения
  AssignOp();
 }
}
```

Пример работы интерпретатора:

```
calc>x=100          //вычислить выражение
100.00              //результат
calc>y=200
200.00
calc>z=y/x
2.00
calc>tabl           .//вывести таблицу символов
0  x   100
1  y   200
2  z   2
calc>end             //завершить работу программы
```

9.4 Интерпретатор простого языка программирования

9.4.1 Описание простого алгоритмического языка

Рассмотрим более сложный пример реализации интерпретатора. Расширим язык арифметических выражений в простой алгоритмический язык, который будет содержать операторы: цикла, присваивания, печати, условного и составного. Ниже приведена грамматика этого алгоритмического языка :

1 ***<Statement>*** => { ***<Statements>***

2 ***<Statements>=> <Statement> <Statements>***
3 ***<Statements>*** => }
4 ***<Statement>*** => ***if*** (***<Express>***) ***<Statement> <Else>***
5 ***<Else> => else <Statement>*** | ***$***
6 ***<Statement>*** => ***while***(***<Express>***) ***<Statement>***
7 ***<Statement>*** => ***<Ident> = <Express>***
8 ***<Express> => print <ListExpress>***
9 ***<ListExpress> => <Express> <ListExpress>***
10 ***<ListExpress> => ,<Express>< ListExpress>***
11 ***<ListExpress>=> $***
12
13 ***<Express>=> <ExpressTest> < <Express>***
14 ***<Express>=> <ExpressTest> > <Express>***
15 ***<Express>=> <ExpressTest>***
16
17 ***<ExpressTest>=> <ExpressAdd> + <ExpressTest>***
18 ***<ExpressTest>=> <ExpressAdd> — <ExpressTest>***
19 ***<ExpressTest>=> <ExpressAdd>***
20
21 ***<ExpressAdd>=> <ExpressMult> * <ExpressAdd>***
22 ***<ExpressAdd>=> <ExpressMult> / <ExpressAddt>***
23 ***<ExpressAdd>=> <ExpressMult>***
24
25 ***<ExpressMult>=>*** (***<ExpressTest>***)
26 ***<ExpressMult>=> sin <ExpressFunc>***
27 ***<ExpressMult>=> <Ident>***
28 ***<ExpressMult>=> <Const>***

Анализ грамматики показывает, что правила с номерами 17–26 описывают арифметические выражения. Кроме того правила 13–15 относятся также к классу арифметических выражений и описывают операции отношения. Правила 1–11 описывают операторы алгоритмического языка. Составной оператор необходим для объединения нескольких операторов (правила 1–3). Условный оператор (правила 4–6) организует ветвления в программе. Оператор цикла (правило 6) предназначен для многократного выполнения некоторой последовательности операторов. Оператор печати обеспечивает печать строк символов и значения переменных.

Ниже приведены два примера программ на простом алгоритмическом языке:

Пример № 1. Вычисление арифметической прогрессии

```
{
 print "sum = 1+2+3+...+500"
 i=0
 while(i<500) {
  sum=sum+i
  i=i+1
 }
 print "sum=",sum
}
```

Пример № 2. Вычисление корня квадратного уравнения

```
{
 a=1 b=-7 c=10
 print "ax^2+bx+c=0"
 d=b*b-4*a*c
 if(d<0) print \" D<0
  else {
  x1=(-1*b+sqrt(d))/(2*a)
  x2=(-1*b-sqrt(d))/(2*a)
  print "x1=",x1,\"x2=",x2
  }
}
```

9.4.2 Реализация интерпретатора

Для реализации воспользуемся интерпретатором арифметических выражений. И добавим несколько дополнительных функций. Прежде всего это служебные функции skipchar и eof. Описание функций приведено ниже.

```
//пропуск символа во входной строке
void skipchar() { if(*ch!='\0') ch++; }

//проверка достижения конца  входной строки
int eof() { return (*ch=='\0');  }
```

Добавим функцию ExpressTest, которая производит разбор выражения с операциями отношения (больше '>', меньше '<', равно '==', неравно '!= '). Кроме того, в каждую функцию, которая производит разбор и вычисление, вводится параметр run. Этот параметр необходим для указания выполнения или не выполнения вычислений. Если run равен 1, то вычисления производятся. Если нет, то не производятся.

Поскольку интерпретатор последовательно производит разбор исходного теста программы, то необходимо задать механизм, когда вычисления производятся в зависимости от логики программы. Поэтому введен параметр run, который указывает, когда надо производить вычисления, а когда не надо. Параметр run устанавливается в следующих случаях: при первоначальном запуске, при разборе условного оператора и оператора цикла.

Ниже приведен текст функции ExpressTest.

```
void ExpressTest(int run) { // вычисление выражения для операций отношения
 double f1, f2;
 ExpressAdd(run);            //вычислить значение первого операнда
 while(1) {
  if(match(">")){            //если больше
    ExpressAdd(run);         //вычислить значение для второго операнда
    if(run){
     f2=Pull(); f1=Pull();   //взять значения слагаемых из стека
     Push(f1>f2);            //положить сумму в стек
     }
   }
 else
  if(match("<")) {     //если меньше
    ExpressAdd(run);    //взять значение второго операнда
    if(run){
     f2=Pull(); f1=Pull(); //взять значения из стека
     Push(f1<f2);          //положить разность в стек
     }
   }
  else
  if(match("==")) {       //если это равно
    ExpressMult(run);    //взять значение второго операнда
    if(run){
     f1=Pull(); f2=Pull(); //взять значения из стека
     Push(f2==f1);          //положить разность в стек
```

```
    }
   }
  if(match("!=")) {        //если не равно
    ExpressMult(run);    //взять значение второго операнда
    if(run){
     f1=Pull(); f2=Pull(); //взять значения из стека
     Push(f2!=f1);           //положить разность в стек
     }
   }
  else break;
 }
}
```

Для разбора и выполнения оператора печати используется функция do_print. Текст этой функции приведен ниже. В теле этого оператора записывается список, в котором через запятую перечисляются строки символов или выражения. Строки символов ограниченны кавычками. Для записи служебных символов используется обратная косая черта.

```
void do_print(int run){
 double r;
 while(1){   //цикл пока не выведены все элементы списка печати
  if(match("\"")) {    //это строка символов
   while(!eof()){      //пока не конец входной строки
    if(*ch=='\"') { ch++; break;}   //если это конец строки
    if(*ch=='\\') ch++;             //если это обратная косая черта, то сдвинуть
    if(run) printf("%c",*ch);   //печать
    ch++;                            //следующий символ
    }
  }
  else {
    ExpressTest(run); //разобрать выражение
    if(run){
     r=Pull();
     printf("%g ",r); .//печать значение выражения
     }
   }
  if(match(",")) continue; //продолжить если список еще не пуст
  break; //иначе выход
  }
 printf("\n");
```

```
}
```

Функция разбора условного оператора и выполнения. Правила выполнения просты: если run установлен в 1, то разобрать и выполнить выражение условия. Если результат выражения не равен нулю, то разобрать и выполнить операторы, следующие за закрывающей круглой скобкой. В противном случае производится просто разбор. Если есть ключевое слово else, то в зависимости от результата r, производится или не производятся вычисления, разбор производится в любом случае.

```
void do_if(int run){ //разбор и выполнение оператора if
 double r;
 if(match("(")){   //разобрать и вычислить условное выражение
  ExpressTest(run);
  match(")");
  if(run){  //если выполнение установлено
    r=Pull();
    if(r!=0)  Statement(1); else Statement(0); //выполнить или не выполнить в
зависимости от результата
    }
  else Statement(run); //разобрать без выполнения

  if(match("else")){ //если есть else
    if(run) { //выполнение включено
     if(r==0) Statement(1);  //результат равен нулю, то выполнить оператор
    else Statement(0); //иначе провести разбор без выполнения
    }
   else Statement(run); //провести разбор
  }
 }
}
```

Функция разбора и выполнения оператора цикла. Если параметр run установлен в ноль, то производится синтаксический разбор оператора цикла. В противном случае производится разбор и выполнение. Первоначально вычисляется условное выражение в скобках, далее в зависимости от значения тело цикла разбирается и выполняется, или только разбирается. В любом случае выход из цикла выполнения производится после разбора тела цикла. Особую роль играет локальная переменная chv, которая хранит значение указатель на текущий символ при входе в функцию. После разбора и выполнения тела цикла необходимо восстановить текущий указатель на на-

чало цикла, это и обеспечивает переменная chv. Поскольку эта переменная локальная, то возможно рекурсивное использование функции do_while. Рекурсия будет организована если в исходном тексте программы будет несколько вложенных операторов цикла.

```
void do_while(int run){
 double r;
 char *chv=ch;
 while(1){
  if(match("(")){  //разобрать условие
    ExpressTest(run);
    match(")");
    if(run){       //выполнение включено
    r=Pull();
    if(r!=0){
       Statement(1); //выполнить тело цикла
       ch=chv;       //продолжить выполение цикла
       }
      else{          //условие не выполенено
       Statement(0); //разобрать тело цикла без выполнения
       break;        //выход из цикла
      }
    }
  else{              //разобрать тело цикла без выполнения
   Statement(run);
   break;
    }
   }//endif
  }//endwhile
}
```

Функция разбора и выполнения составного оператора.

```
void do_compaund(int run){
  while(!eof()){
   if(match("}")) break;
   Statement(run);
  }
}
```

Функция разбора всех операторов алгоритмического языка

```
void Statement(int run){
 if(match("if")) do_if(run); //разобрать и выполнить условный оператор
else
 if(match("while")) do_while(run); //разобрать и выполнить оператор цикла
else
 if(match("{")) do_compaund(run); //разобрать и выполнить составной опера-
тор
else
 if(match("print")) do_print(run); //разобрать и выполнить оператор печати
else
 AssignOp(run); //разобрать и выполнить оператор присваивания
}
```

Рассмотрим несколько примеров работы интерпретатора.

1. Строка записи программы для вычисления арифметической прогрессии

```
char prog0[] = "{\
 print \"sum = 1+2+3+...+500\"\
 i=0\
 while(i<500) {\
   sum=sum+i\
   i=i+1\
   }\
  print \"sum=\",sum\
 }";
```

2. Строка записи программы вычисления корня квадратного

```
char prog1[] = "{\
 a=1 b=-7 c=10\
 print \"ax^2+bx+c=0\"\
 d=b*b-4*a*c\
 if(d<0) print \" D<0 \"\
  else{\
   x1=(-1*b+sqrt(d))/(2*a)\
   x2=(-1*b-sqrt(d))/(2*a)\
   print \"x1=\",x1,\"x2=\",x2\
   }\
```

```
 }";
```

Программа для ввода команды или программы и вызова интерпретатора

```
void main() {
 TopVar=0;          //инициализация стека и таблицы переменных
 TopStack=0;
 while(1) {
  printf("end   - exit of programm\n\
tabl  - print table of identifair\n\
prog0 - run program <sum=1+2+3+...+500>\n\
prog1 - run program <ax^2+bc+c=0>\n\
run input string\n");
  printf("interpet>");   //вывод подсказки
  gets(in_line);             //ввод строки символов
  ch=&in_line[0];        //инициализация указателя разбора
   if(match("end"))       //если это "end", то завершение работы
     break;
  else
  if(match("tabl"))       //если это "tabl", то вывести таблицу символов
     PrintTablVar();
  else
  if(match("prog0")){   //выполнение первой программы
    strcpy(in_line,prog0);
    ch=&in_line[0];
    Statement(1);
   }
  else
  if(match("prog1")){   //выполнение второй программы
    strcpy(in_line,prog1);
    ch=&in_line[0];
    Statement(1);
   }
  else                    //иначе вычислить значение выражения
    Statement(1);
 }
}
```

Задания и упражнения

1. Ввести в алгоритмический язык обработку различного класса объектов: векторов, матриц, списков, структур.
2. Вести в алгоритмический язык оператор ввода.
3. Ввести понятие функций и подпрограмм.
4. Ввести режим пошагового выполнения программы.
5. Ввести расширенную диагностику ошибок.

9.5 Компилятор усеченного языка программирования Си

Построение компилятора для реального, хотя и усеченного, языка программирования является достаточно сложным делом. В литературе [ ] опубликован текст компилятора RatC для усеченного языка программирования Си. Исходный код этого компилятора имеется в файле . Дадим краткое описание этого компилятора.

Этот компилятор генерирует ассемблерный код для некоторого виртуального микропроцессора, похожего на микропроцессор Intel 8080. В нем имеется четыре регистра:

PC — счетчик команд, SP — указатель стека, P — первый регистр данных, S — второй регистр данных.

Система команд следующая:

ldir.b	<метка>	загрузка данных записанных в байте по адресу <метка> в регистр P
ldir.w	<метка>	загрузка данных записанных в слове(16 разрядов по адресу <метка> в регистр P
addr	<значение>	P=SP+значение
sdir.b	<метка>	сохранение байта их регистра P в байт по адресу <метка>
sdir.w	<метка>	сохранение слова их регистра P в слово по адресу <метка>
sind.b		(S)=P.b пересылка байта из P по адресу заданному в S
sind.w		(S)=P (пересылка слова из P по адресу заданному в S)
lind.b		считывание байта в регистр P, по адресу записанному в P
lind.w		считывание слова в регистр P, по адресу записанному в P
call	<метка>	вызов подпрограммы (адрес возврата заносится в стек)
scall		вызов подпрограммы из вершины стека (адрес воз-

		врата заносится в стек)
ujump	<метка>	безусловный переход
fjump	<метка>	переход если P==0
modstk	<значение>	SP=SP+значение
swap		обмен содержимого P и S
limm	<значение>	P=занчение
push		содержимое P заслать в стек
pop		вытащить из стека и записать в S
xchange		обмен значениями P и вершины стека
return		возврат из подпрограммы
scale	<значение>	умножить содержимое P на занчение
add		P=S+вершина стека
sub		P=S-P
mult		P=S*P
div		P=S/P остаток от деления в S
mod		S=S/P остаток в P
or		P=P|S поразрядное или
xor		P=P^S поразрядное исключающее или
and		P=P&S поразрядной И
asr		арифметический сдвиг вправо S на число бит записанных в P, результат в P
asl		арифметический сдвиг влево S на число бит записанных в P, результат в P
neg		поразрядная инверсия P
inc		P=P+1
dec		P=P-1
testeq		если S равно P то P=1 иначе P=0
testne		если S неравно P то P=1 иначе P=0
testlt		если S меньше P то P=1 иначе P=0
testle		если S меньше или равно P то P=1 иначе P=0
testgt		если S больше P то P=1 иначе P=0
testge		если S больше или равно P то P=1 иначе P=0
testult		если S меньше P то P=1 иначе P=0 (без знака)
testule		если S меньше или равно P то P=1 иначе P=0 (без знака)
testugt		если S больше P то P=1 иначе P=0 (без знака)
testuge		если S больше или равно P то P=1 иначе P=0 (без знака)

Перенос и использование компилятора RatC

Особенность этого транслятора — транслятор написан на том же усеченном Си, что является входным для самого транслятора. Это означает, что транслятор можно переносить с одной вычислительной среды в другую. Например, если имеется транслятор RatC и его исходный текст для вычислительной системы A, необходимо получить транслятор для вычислительной системы B, то можно это получить следующим образом:

1) изменяем исходный текст RatC(A) для вычислительной системы B, получаем исходный текст RatC(B);
2) компилируем исходный текст RatC(B) на компиляторе RatC(A), получаем компилятор RatC(B), работающий на вычислительной системе A;
3) компилируем исходный текст RatC(B) на компиляторе RatC(B) для вычислительной системы A, получаем компилятор RatC(B), работающий на вычислительной системе B.

Эта схема работает при условии, что системные библиотеки вычислительных систем A и B совпадают.

Использовать компилятора RatC можно по нескольким направлениям:

1. Написать виртуальную машину, моделирующую систему команд RatC.
2. Переделать RatC для системы команд заданного процессора.
3. Переделать RatC для конкретного ассемблера.

9.6 Виртуальные машины

9.6.1 Основные понятия

Под виртуальной машиной (ВМ) будем подразумевать программную реализацию модели некоторой вычислительной системы. Практически все известные системы программирования можно рассматривать как виртуальные машины, поскольку часть операций моделируются, т.е. для них написан программный код. При построении ВМ необходимо рассмотреть следующие вопросы:

– структура ВМ;
– множество элементарных данных;
– множество элементарных операций;
– управление последовательностью действий;
– управление данными;
– управление памяти.

Структура ВМ определяется наличием интерпретатора команд и специальных блоков памяти, где хранятся данные, программы и вспомогательная информация. Множество элементарных данных и операций определяется из анализа состава функций и задач некоторой предметной области.

Управление последовательностью действий включает два аспекта: управление последовательностью операций и управление вызова подпрограмм. Вопросы управления данными подразумевает рассмотрение способов хранения и передачи данных между подпрограммами.

При создании вычислительного процесса данные могут накапливаться, изменяться и удаляться, поэтому необходимо рассмотреть также вопросы управления памятью.

При построении ВМ для данной предметной области необходимо иметь состав функций, который может быть выявлен в ходе проведения системного анализа с целью построения модели предметной области. Если этот состав конечен и полон, т.е. все множество алгоритмов в данной предметной можно получить из данного состава функций и при этом множество этих алгоритмов достаточно велико или бесконечно, то необходимо строить ВМ.

9.6.2 Простая виртуальная машина

Виртуальная машина состоит из интерпретатора команд, буфера для хранения кода программы и буфера для хранения данных.

Интерпретатор выполняет следующие команды:

Мнемоника команды	Операнд 1	Операнд 2	Операнд 3	Комментарий
in	addr			ввод значения
out	addr			вывод значения
add	addr1	addr2		сложить
sub	addr1	addr2		вычесть
mul	addr1	addr2		умножить
div	addr1	addr2		разделить
jump	addr1			безусловный переход
grt	addr1	addr2		больше
lst	addr1	addr2		меньше
bra	addr1			условный переход
stop				
test	addr			равно нулю
mov	addr1	addr2		копировать данные

Мнемоника команды	Операнд 1	Операнд 2	Операнд 3	Комментарий
iload	addr1	addr2	addr3	загрузить данные косвенно
isave	addr1	addr2	addr3	сохранить данные косвенно
inc	addr			инкремент
dec	addr			декремент

Каждая команда имеет фиксированную длину и состоит из кода операции и операндов. Код операции это число идентифицирующее операцию. Код операции занимает один байт. Операнд указывает на ячейку памяти, где хранится необходимая информация или куда будет занесен результат операции. В команде может быть несколько операндов. Например, для команды «ADD» необходимо указать адрес первого слагаемого и адрес второго слагаемого. Результат сложения будет занесен по адресу первого слагаемого. Все операнды имеют размер одного байта. Тогда всю программу можно размесить в байтовом массиве.

Поскольку адрес записывается в один байт то размер буферов для данных и программы не должен превышать 256. Алгоритм интерпретатора команд достаточно простой:

1) счетчик команд (pc) сбрасывается в ноль;
2) читается очередной код команды (code=prog[pc]);
3) по коду происходит переход на обработку заданной команды;
4) производится выборка операндов;
5) выполняется операция, запоминается результат.
6) корректируется счетчик команд в соответствии с количеством операндов;
7) происходит переход на выполнение шага 2.

9.6.3 Реализация простой виртуальной машины

Таблица кодов команд виртуальной машины

```
#define   IN      1       //ввод числа
#define   OUT     2       //вывод числа
#define   ADD     3       //сложить
#define   SUB     4       //вычесть
#define   MULT    5       //умножить
#define   DIV     6       //разделить
#define   JUMP    7       //безусловный переход
#define   GRT     8       //проверка на больше
#define   LST     9       //проверка на меньше
```

```
#define   BRA      10        //перейти если флаг установлен
#define   STOP     11        //останов
#define   TEST     12        //проверка на ноль
#define   MOV      13        //переслать данные
#define   ILOAD    14        //косвенная загрузка данных
#define   ISAVE    15        //косвенное соранение данных
#define   INC      16        //увеличить на единицу
#define   DEC      17        //уменьшить на единицу

#define SIZEMEMORY 256   //размер памяти под данные
#define DEBUG_SVM          //работа программы в отладочном режиме
int Mem[SIZEMEMORY];     //память по данные
```

Входные параметры :

prog — массив, содержащий программу (последовательность команд);

Mem — массив, содержащий данные.

Основные переменные:

com — код команды;

Addr, Addr2, Addr3 — вспомогательные переменные для хранения операндов текущей команды;

flag — переменная, хранящая флаг перехода;

pc — счетчик команд.

```
int SimpleVirtualMachine(unsigned char *prog,int Mem[]){
  int com, //код команды
      Addr,Addr2,Addr3, //вспомогательные ячейки для хранения адресов
      flag=0,                //флаг перехода
      pc=0;                  //счетчик команд (program counter)
  while(1) {                 //цикл выполнения программы
#ifdef DEBUG_SVM   //для отладки
   PrintMemory();
   PrintCommand(&prog[pc]);
   if(getch()==27) exit(0); //если нажата ESC завершить программу
#endif
   com=prog[pc]; //выделяем код операции
   switch (com){
  case IN://добавляем 1, т.к. вначале pc
         //указывает на IN, а нам необходимо само число
         Addr=prog[pc+1];   //ввод числа
```

```
        printf("svm_input>");
        scanf("%d",&Mem[Addr]);
        pc=pc+2;  //смещаем счетчик команд
        break;
  case OUT: //вывод числа
       Addr=prog[pc+1];
       printf("svm_ouput>%d\n",Mem[Addr]);
       pc=pc+2;  //смещаем счетчик команд
       break;
  case ADD:      //сложить два числа
       Addr=prog[pc+1];
       Addr2=prog[pc+2];
       //производим сложение и результат записываем по адресу первого эле-
мента
       Mem[Addr]=Mem[Addr]+Mem[Addr2];
       pc=pc+3;
       break;
  case SUB: //вычитание
       Addr=prog[pc+1];
       Addr2=prog[pc+2];
       Mem[Addr]=Mem[Addr]-Mem[Addr2];
       pc=pc+3;
       break;
  case MULT: //умножение
       Addr=prog[pc+1];
       Addr2=prog[pc+2];
       Mem[Addr]=Mem[Addr]*Mem[Addr2];
       pc=pc+3;
       break;
  case DIV: //деление
       Addr=prog[pc+1];
       Addr2=prog[pc+2];
       Mem[Addr]=Mem[Addr]/Mem[Addr2];
       pc=pc+3;
       break;
  case JUMP://безусловный переход
        pc=prog[pc+1];
        break;
  case TEST: //проверка на ноль и установка флага
        Addr=prog[pc+1];
        if (Mem[Addr]==0) flag=1;
```

```
        else flag=0;
        pc=pc+2;
        break;
case GRT: //проверка a>b
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        if (Mem[Addr]>Mem[Addr2]) flag=1;  //установка флага
        else flag=0;                       //сброс флага
        pc=pc+3;
        break;
case LST: //проверка a<b
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        if (Mem[Addr]<Mem[Addr2]) flag=1;
        else flag=0;
        pc=pc+3;
        break;
case BRA: //условный переход
        Addr=prog[pc+1];
        if(flag) pc=Addr;
       else pc=pc+2;
        break;
case INC: //Инкримент
        Addr=prog[pc+1];
        Mem[Addr]++;
        pc=pc+2;
        break;
case DEC: //Декримент
        Addr=prog[pc+1];
        Mem[Addr]++;
        pc=pc+2;
        break;
case MOV: //переслать данные
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        Mem[Addr]=Mem[Addr2];
        pc=pc+3;
        break;
case ILOAD: //косвенная загрузка данных
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
```

```
        Addr3=prog[pc+3];
        Mem[Addr]=Mem[Addr2+Mem[Addr3]];
        pc=pc+4;
        break;
  case ISAVE: //косвенное сохранение данных
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        Addr3=prog[pc+3];
        Mem[Addr2+Mem[Addr3]]=Mem[Addr];
        pc=pc+4;
        break;
  case STOP: return 1;
  default: printf("Error command\n"); return -1;
   }
 }
}


//печать первых 10 ячеек памяти
void PrintMemory(){
 printf("dump memory ----------\n");
 for(int i=0; i<10; i++)
  printf("%2d %5d\n",i,Mem[i]);
}

//вывод команды на терминал
int PrintCommand(unsigned char *prog){
  int Addr,Addr2,Addr3;
  int pc=0;
  switch (*prog){
  case IN:
        Addr=prog[pc+1];   //ввод числа
        printf(" IN %3d\n",Addr);
        break;
  case OUT: //вывод числа
        Addr=prog[pc+1
        printf("OUT %3d\n",Addr);
        break;
  case ADD: //сложить два числа
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
```

```
        printf("ADD %3d, %3d\n",Addr,Addr2);
       break;
case SUB: //вычитание
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        printf("SUB %3d, %3d\n",Addr,Addr2);
        break;
case MULT: //умножение
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        printf("MULT %3d, %3d\n",Addr,Addr2);
        break;
       break;
case DIV: //деление
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        printf("DIV %3d, %3d\n",Addr,Addr2);
        break;
       break;
case JUMP://безусловный переход
        Addr=prog[pc+1];   //ввод числа
        printf("JUMP %3d\n",Addr);
        break;
case TEST: //проверка на ноль и установка флага
        Addr=prog[pc+1];   //ввод числа
        printf("TEST %3d\n",Addr);
        break;
case GRT: //проверка a>b
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        printf("GRT %3d, %3d\n",Addr,Addr2);
        break;
case LST: //проверка a<b
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        printf("LST %3d, %3d\n",Addr,Addr2);
        break;
case BRA: //условный переход
        Addr=prog[pc+1];
        printf("BRA %3d\n",Addr);
        break;
```

```
  case INC: //Инкремент
        Addr=prog[pc+1];
        printf("INC %3d\n",Addr);
        break;
  case DEC: //Декремент
        Addr=prog[pc+1];
        printf("DEC %3d\n",Addr);
        break;
  case MOV: //переслать данные
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        printf("MOV %3d, %3d\n",Addr,Addr2);
        break;
  case ILOAD: //косвенная загрузка данных
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        Addr3=prog[pc+3];
        printf("ILOAD %3d, %3d, %3d\n",Addr,Addr2,Addr3);
        break;
  case ISAVE: //косвенное сохранение данных
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        Addr3=prog[pc+3];
        printf("ILOAD %3d, %3d, %3d\n",Addr,Addr2,Addr3);
        break;
  case STOP: printf("STOP\n"); return 1;
  default: printf("Error command\n"); return -1;
  }
 }

//Пример1
//Распределение памяти
#define X   0
#define Y   1
#define SUM 2

//программа для виртуальной машины (SVM)
//ввод x, y; вычисдение sum=x+y, вывод sum
unsigned char prog1[]= {
  IN, X,       //ввести значение для X
  IN, Y,       //ввести y
```

```
 MOV, SUM, X,  //переслать значение X в SUM
 ADD, SUM, Y,  //найти сумму
 OUT, SUM,     //вывести сумму
 STOP          //останов
};


//Пример2
//распределение памяти
#define COUNT  0
#define NMAS   1
#define SMAS   2
#define CURE   3
#define MAS    4
#define BEG    0


//программа2 для виртуальной машины (SVM)
//нахождение суммы элементов массива
// для(i=0; i<n; i++) sum=sum+mas[i]; вывод(sum)
unsigned char prog2[]= {
 ILOAD, CURE,MAS,COUNT,  //cure=mas[count]
 ADD,  SMAS,CURE,        //smas=smas+cure
 INC,  COUNT,            //count=cur+1
 LST,  COUNT,NMAS,       //count<nmas?
 BRA,  BEG,              //переход по BEG
 OUT,  SMAS,             //вывод суммы(smas)
 STOP                    //останов
};

void SetMemory(){  //начальная установка памяти
 Mem[COUNT]=0;     //установка счетчика элементов массива
 Mem[NMAS]=50;     //количество элементов в массиве
 Mem[SMAS]=0;      //начальное значение суммы
 for(int i=0; i<50; i++) Mem[MAS+i]=i+1;  //начальные значения для массива
}

void main(){
 SimpleVirtualMachine(prog1,Mem);   //выполение первой программы
 //SetMemory();
 //SimpleVirtualMachine(prog2,Mem);     //выполение второй программы
```

```
 getch();
}
```

9.7 Система — виртуальная машина + транслятор

9.7.1 Эволюция программных систем

Как правило, сложная программная система (ПС) претерпевает несколько стадий своего развития:

1) первоначальная версия;

2) конфигурационная версия;

3) инструментальная версия.

Первоначальная версия системы — это жесткая программа, которая ориентирована на выполнение конкретных функций, при этом модификация функций не предполагается.

Конфигурационная версия предполагает существование специального файла конфигураций, в котором могут быть записаны различные варианты функций и параметров ПС. Как правило, в этом файле записывается последовательность строк, следующей структуры:

<параметр> = <значение>,

где <параметр> некоторый параметр ПС или название функции;

<значение> — это числовая константа или строка символов, означающая некоторый вариант функции или параметра.

При таком подходе возможно подстраивать ПС под конкретные нужды потребителей, изменяя значения в конфигурационном файле.

Инструментальная версия ПС является дальнейшим развитием конфигурационной ПС, и предполагает наличие некоторого языка, на котором описывается вариант реализации ПС. Этот язык является предметно ориентированным и, в зависимости от особенностей предметной области, может быть декларативным или процедурным. Теперь использование ПС состоит в два этапа: создание конкретной версии ПС и выполнение данной конкретной ПС. Таким образом, инструментальную ПС можно разделить на две части: системная часть, отвечающую за реализацию языка и проблемную часть, в которой содержатся программы реализующие алгоритмы обработки в данной предметной области.

Системная часть производит анализ программы (описания конкретной варианта ПС) и генерацию конкретного варианта ПС. Если предметный язык процедурного типа, то создается транслятор, если декларативного — то генератор. Проблемная часть может быть также реализована несколь-

ми способами: библиотека подпрограмм, интерпретатор некоторого внутреннего кода. Библиотека подпрограмм создается в тех случаях, когда состав функций данной предметной области может расширяться. Тогда с внесением новой подпрограммы меняется и системная часть. В настоящее время сложные программные системы строятся по принципу «транслятор — виртуальная машина». Например, так реализована система программирования для интернета — Java.

9.7.2 Описание языка

Синтаксис языка описывается следующей грамматикой:
<Программа> => ***program*** <операторы программы>
<Операторы программы> => <Оператор> < Операторы программы >
<Операторы программы>=> ***endprog***
<Оператор>=> ***if*** <Выражение> ***then*** <Операторы> ***endif***
<Оператор>=> ***if*** <Выражение> ***then*** <Операторы> ***else*** <Операторы> ***endif***
<Оператор>=> ***cycle*** <Выражение> <Операторы> ***endcycle***
<Оператор>=> <Идентификатор>:=<Выражение>
<Оператор>=> output <Список выражений>
<Оператор>=> input < Идентификатор >
<Оператор>=> ;
<Операторы> => <Оператор> < Операторы>

Комментарий в программе оформляется следующим образом: пишется два подряд символа '-' и далее до конца строки все символы воспринимаются как комментарий. Например,

```
-- Это комментарий к программе;
-- увеличить счетчик
```

Переменные в программе не объявляются, тип автоматически определяется в зависимости от значения, которое будет присвоено. Типы данных следующие: целый, вещественный, строка символов, точка. Более подробно смотри реализацию.

Примеры программ

Пример № 1. Нахождение суммы чисел натурального ряда. Первые две строки комментарий к программе. Затем устанавливаются значения двух переменных i — текущее число натурального ряда и s — сумма. Затем

организуется цикл подсчета суммы. В цикле выводится значение текущего натурального числа. По окончанию цикла выводится значение суммы.

```
--this program count sum
--15.01.04
 program
 i:=1; s:=0;
 cycle(i<5)
    s:=s+i;
    i:=i+1;
   output i;
   endcycle
 output s;
endprog
```

Пример № 2. Нахождение значения n! (факториал).

```
program
 --factorial(10)
 i:=1; s:=1;
 cycle(i<10)
  s:=s*i; i:=i+1;
  output i;
 endcycle
 output s;
endprog
```

9.7.3 Стековая виртуальная машина

Стековая виртуальная машина является удобным инструментом в сочетании с транслятором, основанном на методе рекурсивного спуска. В стековой ВМ операнды операции берутся из стека и результаты операций также заносятся в стек (см. раздел использование стека для вычисления арифметических выражений). Стек используется для хранения промежуточных результатов. Сами объекты хранятся в специальной памяти и есть команду Push и Pull — которые вставляют в стек объект из памяти и выталкивают объект из стека и запоминают его адресу в памяти.

Ниже приведено описание заголовочного файла с описанием параметров стековой виртуальной машины.

Параметры размера памяти под объекты и стека

```
#define SIZEMEMORY 256
#define SIZESTACK  256
```

Ключ установки отладочного режима

```
//#define DEBUG_SVM
```

Описание типа объекта

```
#define INT          1     //целый
#define DOUBLE  2      //вещественный
#define STRING    3      //строка
#define POINT      4      //точка
```

Команды виртуальной машины

#define	IN	1	//ввод числа
#define	OUT	2	//вывод числа
#define	ADD	3	//сложить
#define	SUB	4	//вычесть
#define	MULT	5	//умножить
#define	DIV	6	//разделить
#define	JUMP	7	//безусловный переход
#define	GRT	8	//проверка на больше
#define	LST	9	//проверка на меньше
#define	BEQ	10	//перейти если флаг установлен
#define	STOP	11	//останов
#define	EQU	12	//проверка на равно
#define	MOV	13	//переслать данные
#define	ILOAD	14	//косвенная загрузка данных
#define	ISAVE	15	//косвенное соранение данных
#define	INC	16	//увеличить на единицу
#define	DEC	17	//уменьшить на единицу
#define	PUSH	18	//втолкнуть в стек
#define	PULL	19	//вытащить из стека

#define	TOP	20	//взять вершину стека
#define	BNE	21	//не равно

Описание структуры точки

```
typedef struct PointTag {
 int x;
 int y;
} Point;
```

Описание объекта

```
typedef struct ObjectTag {

  unsigned char type;     //тип хранимого объекта
  union {                 //объединение различных типов
  int int_number;          //целое число
  double float_number;  //вещественное число
  char string[20];         //строка
  Point p;                  //структура ТО
  } m;
} Object;    //это все может храниться в стеке
```

Стек для хранения объектов

```
Object Stack[SIZESTACK];
int Topp;                                  //указатель стека
Object Mem[SIZEMEMORY];    //память по данные
```

Функция заносит значение объекта в стек.

```
int Push(Object *obj){
  if(Topp<SIZESTACK){
   Stack[Topp++]=*obj;
   return 1;          //нормальное завершение
  }
  return -1;          //стек переполнен
}
```

Функция выталкивает значение объекта из стека

```
int Pull(Object *obj){ //забрать из стека
  if(Topp>0){
   *obj=Stack[--Topp];
   return 1;          //нормальное завершение
  }
  return -1;          //стек пуст
}
```

Функция взятие значения из вершины стека

```
int Top(Object *obj){
  if(Topp>0){
   *obj=Stack[Topp-1];
   return 1;          //нормальное завершение
  }
  return -1;          //стек пуст
}
```

Функция проверки отношения a>b . Отношение проверяется в зависимости от типа. Если отношение выполняется, то результат равен единице. В противном случае 0.

```
int Grt(Object *op1, Object *op2){
 char buf[20];
 if(op1->type==INT && op2->type==INT ){
   if(op1->m.int_number > op2->m.int_number) return 1;
   else return 0;
   }
 else
  if(op1->type==DOUBLE && op2->type==INT ){
   if(op1->m.float_number > op2->m.int_number) return 1; //
   else return 0;
   }
 else
   if(op1->type==INT && op2->type==DOUBLE ){
   if(op1->m.int_number>op2->m.float_number) return 1;
   else return 0;
   }
 else
   if(op1->type==DOUBLE && op2->type==DOUBLE ){
     if(op1->m.float_number>op2->m.float_number) return 1;
```

```
    else return 0;
   }
 else
   if(op1->type==STRING && op2->type==STRING ){
    if(strcmp(op1->m.string,op2->m.string)>0) return 1;
    else return 0;
   }
  else return 0; //можно дальше продолжить
}
```

Функция проверки отношения a<b . Отношение проверяется в зависимости от типа. Если отношение выполняется, то результат равен единице. В противном случае 0.

```
int Lst(Object *op1, Object *op2){
 char buf[20];
 if(op1->type==INT && op2->type==INT ){
   if(op1->m.int_number < op2->m.int_number) return 1;
   else return 0;
   }
 else
  if(op1->type==DOUBLE && op2->type==INT ){
   if(op1->m.float_number < op2->m.int_number) return 1; //
   else return 0;
   }
 else
   if(op1->type==INT && op2->type==DOUBLE ){
   if(op1->m.int_number < op2->m.float_number) return 1;
   else return 0;
   }
 else
   if(op1->type==DOUBLE && op2->type==DOUBLE ){
     if(op1->m.float_number < op2->m.float_number) return 1;
     else return 0;
   }
 else
   if(op1->type==STRING && op2->type==STRING ){
    if(strcmp(op1->m.string,op2->m.string)<0) return 1;
    else return 0;
   }
  else return 0; //можно дальше продолжить
```

```
}

//равно  a==b
int Equ(Object *op1, Object *op2){
 char buf[20];
 if(op1->type==INT && op2->type==INT ){
   if(op1->m.int_number == op2->m.int_number) return 1;
   else return 0;
   }
 else
  if(op1->type==DOUBLE && op2->type==INT ){
   if(op1->m.float_number == op2->m.int_number) return 1; //
   else return 0;
   }
 else
   if(op1->type==INT && op2->type==DOUBLE ){
   if(op1->m.int_number == op2->m.float_number) return 1;
   else return 0;
   }
 else
   if(op1->type==DOUBLE && op2->type==DOUBLE ){
     if(op1->m.float_number == op2->m.float_number) return 1;
     else return 0;
   }
 else
   if(op1->type==STRING && op2->type==STRING ){
    if(strcmp(op1->m.string,op2->m.string)==0) return 1;
    else return 0;
   }
  else return 0; //можно дальше продолжить
}

//функция сложения с учетом типа операнда
void Add(Object *op1, Object *op2, Object *res){
 char buf[20];
 if(op1->type==INT && op2->type==INT ){
   res->type=INT;
   res->m.int_number=op1->m.int_number+op2->m.int_number; //int=int+int
   }
 else
  if(op1->type==DOUBLE && op2->type==INT ){
```

```
  res->type=DOUBLE;
  res->m.float_number=op1->m.float_number+op2->m.int_number; //
  }
 else
  if(op1->type==INT && op2->type==DOUBLE ){
  res->type=DOUBLE;
  res->m.float_number=op1->m.int_number+op2->m.float_number;
  }
 else
  if(op1->type==DOUBLE && op2->type==DOUBLE ){
  res->type=DOUBLE;
  res->m.float_number=op1->m.float_number+op2->m.float_number;
  }
 else
  if(op1->type==STRING && op2->type==STRING ){
  res->type=STRING;
  strcpy(res->m.string,op1->m.string);
  strcat(res->m.string,op2->m.string);
  }
 else
  if(op1->type==STRING && op2->type==INT ){
  res->type=STRING;
  strcpy(res->m.string,op1->m.string);
  itoa(op2->m.int_number,buf,10);
  strcat(res->m.string,buf);
  }
 else
  if(op1->type==STRING && op2->type==DOUBLE ){
  res->type=STRING;
  strcpy(res->m.string,op1->m.string);
  sprintf(buf,"%f",op2->m.float_number);
  strcat(res->m.string,buf);
  }
 else ; //можно дальше продолжить
}

//функция вычитания с учетом типа операнда
void Sub(Object *op1, Object *op2, Object *res){
 if(op1->type==INT && op2->type==INT ){
  res->type=INT;
  res->m.int_number=op1->m.int_number-op2->m.int_number; //int=int-int
```

```
  }
 else
  if(op1->type==DOUBLE && op2->type==INT ){
   res->type=DOUBLE;
   res->m.float_number=op1->m.float_number-op2->m.int_number; //
   }
 else
   if(op1->type==INT && op2->type==DOUBLE ){
   res->type=DOUBLE;
   res->m.float_number=op1->m.int_number-op2->m.float_number;
   }
 else
   if(op1->type==DOUBLE && op2->type==DOUBLE ){
   res->type=DOUBLE;
   res->m.float_number=op1->m.float_number-op2->m.float_number;
   }
  else ; //можно дальше продолжить
}

//функция умножения с учетом типа операнда
void Mult(Object *op1, Object *op2, Object *res){
 if(op1->type==INT && op2->type==INT ){
   res->type=INT;
   res->m.int_number=op1->m.int_number*op2->m.int_number; //int=int*int
   }
 else
  if(op1->type==DOUBLE && op2->type==INT ){
   res->type=DOUBLE;
   res->m.float_number=op1->m.float_number*op2->m.int_number; //
   }
 else
   if(op1->type==INT && op2->type==DOUBLE ){
   res->type=DOUBLE;
   res->m.float_number=op1->m.int_number*op2->m.float_number;
   }
 else
   if(op1->type==DOUBLE && op2->type==DOUBLE ){
   res->type=DOUBLE;
   res->m.float_number=op1->m.float_number*op2->m.float_number;
   }
  else ; //можно дальше продолжить
```

```
}

void Div(Object *op1, Object *op2, Object *res){
 if(op1->type==INT && op2->type==INT ){
   res->type=INT;
   res->m.int_number=op1->m.int_number/op2->m.int_number; //int=int/int
    }
 else
  if(op1->type==DOUBLE && op2->type==INT ){
   res->type=DOUBLE;
   res->m.float_number=op1->m.float_number/op2->m.int_number; //
    }
 else
   if(op1->type==INT && op2->type==DOUBLE ){
   res->type=DOUBLE;
   res->m.float_number=op1->m.int_number/op2->m.float_number;
    }
 else
   if(op1->type==DOUBLE && op2->type==DOUBLE ){
   res->type=DOUBLE;
   res->m.float_number=op1->m.float_number/op2->m.float_number;
    }
  else ; //можно дальше продолжить
}

//печать первых 10 ячеек памяти
void PrintMemory(){
 printf("dump memory ----------\n");
 for(int i=0; i<10; i++)
  PrintObjectAddr(i,&Mem[i]);
}

//ввод объекта
void InputObject(Object *res){
 int ch;
 char buf[40];
 while(1){
  printf("Input type (int-'1', float-'2', string-'3', point-'4')\n");
  printf("type>");
  ch=getch();
  printf("%c\n",ch);
```

```
  ch-='0';
  if(ch>0&&ch<5) res->type=ch;
  else {
   puts("Error type\n");
   continue;
   }
  switch(ch){
  case 1: printf("int>");
          gets(buf);
          res->m.int_number=atoi(buf);
          return;
  case 2: printf("float>");
          gets(buf);
          res->m.float_number=atof(buf);
          return;
  case 3: printf("string>");
          gets(buf);
          strcpy(res->m.string,buf);
          return;
  case 4: printf("point>");
          gets(buf);
          return;
    }
  }
}

//печать объекта
void OutputObject(Object *res){
 switch(res->type){
  case 1: printf("out->int>");
          printf("%d\n",res->m.int_number);
          return;
  case 2: printf("out->float>");
          printf("%g\n",res->m.float_number);
          return ;
  case 3: printf("out->string>");
          printf("%s\n",res->m.string);
          return;
  case 4: puts("out->point>");
          return;
    }
```

```
}

//печать объекта для дампа
void PrintObjectAddr(int Addr,Object *res){
 switch(res->type){
  case 1: printf("%03d (int)%d\n",Addr,res->m.int_number);
          return;
  case 2: printf("%03d (float)%g\n",Addr,res->m.float_number);
          return ;
  case 3: printf("%03d (string)%20s\n",Addr,res->m.string);
          return;
  case 4: puts("point>");
          return;
    }
}

//вывод команды на терминал
int PrintCommand(unsigned char *prog){
  int Addr,Addr2,Addr3;
  int pc=0;
  switch (*prog){
  case IN:
          Addr=prog[pc+1];   //ввод числа
          printf("IN   %3d\n",Addr);
          pc+=2;
          break;

  case OUT: //вывод числа
          Addr=prog[pc+1];
          printf("OUT  %3d\n",Addr);
          pc+=2;
          break;
  case PUSH:
          Addr=prog[pc+1];
          printf("PUSH %3d\n",Addr);
          pc+=2;
          break;
  case PULL:
          Addr=prog[pc+1];
          printf("PULL %3d\n",Addr);
          pc+=2;
```

```
        break;
case TOP:
        Addr=prog[pc+1];
        printf("TOP  %3d\n",Addr);
        pc+=2;
        break;
case ADD: //сложить два числа
        printf("ADD\n");
        pc++;
       break;
case SUB: //вычитание
        printf("SUB\n");
        pc++;
        break;
case MULT: //умножение
        printf("MULT\n");
        pc++;
        break;
case DIV: //деление
        printf("DIV\n");
        pc++;
         break;
case JUMP://безусловный переход
        Addr=prog[pc+1];   //ввод числа
        printf("JUMP %3d\n",Addr);
        pc+=2;
        break;
case EQU: //проверка на равно
        printf("EQU\n");
        pc++;
        break;
case GRT: //проверка a>b
        printf("GRT \n");
        pc++;
        break;
case LST: //проверка a<b
        printf("LST \n");
        pc++;
        break;
case BEQ: //условный переход
        Addr=prog[pc+1];
```

```
        printf("BEQ  %3d\n",Addr);
        pc+=2;
        break;
case BNE: //условный переход
        Addr=prog[pc+1];
        printf("BNE  %3d\n",Addr);
        pc+=2;
        break;
case INC: //Инкримент
        Addr=prog[pc+1];
        printf("INC  %3d\n",Addr);
        pc+=2;
        break;
case DEC: //Декримент
        Addr=prog[pc+1];
        printf("DEC  %3d\n",Addr);
        pc+=2;
        break;
case MOV: //переслать данные
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        printf("MOV  %3d, %3d\n",Addr,Addr2);
        pc+=3;
        break;
case ILOAD: //косвенная загрузка данных
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        Addr3=prog[pc+3];
        printf("ILOAD %3d, %3d, %3d\n",Addr,Addr2,Addr3);
        pc+=4;
        break;
case ISAVE: //косвенное сохрание данных
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        Addr3=prog[pc+3];
        printf("ILOAD %3d, %3d, %3d\n",Addr,Addr2,Addr3);
        pc+=4;
        break;
case STOP: printf("STOP\n"); return 0;
default: printf("Error command\n"); return -1;
}
```

```
 return pc;
 }

//программа моделирования стековой виртуальной машины
int StackVirtualMachine(unsigned char *prog,Object Mem[]){
  int com, //код команды
      Addr,Addr2,Addr3,//вспомогательные ячейки для хранения адресов
      pc=0;           //счетчик команд (program counter)
      Object obj1,obj2,obj3,res;
  while(1) {   //цикл выполнения программы
#ifdef DEBUG_SVM   //для отладки
   PrintMemory();
   PrintCommand(&prog[pc]);
   if(getch()==27) exit(0); //если нажата ESC завершить программу
#endif
   com=prog[pc]; //выделяем код операции
   switch (com){
  case IN: Addr=prog[pc+1];
           InputObject(&Mem[Addr]);
           pc=pc+2;  //смещаем счетчик команд
             break;
  case OUT: //вывод числа
           Addr=prog[pc+1];
           OutputObject(&Mem[Addr]);
           pc=pc+2;  //смещаем счетчик команд
             break;
  case PUSH:
             Addr=prog[pc+1];
             Push(&Mem[Addr]);
             pc=pc+2;
             break;
  case PULL:
             Addr=prog[pc+1];
             Pull(&Mem[Addr]);
             pc=pc+2;
             break;
  case TOP:
             Addr=prog[pc+1];
             Top(&Mem[Addr]);
             pc=pc+2;
             break;
```

```
case ADD: //сложить два числа
        Pull(&obj1);
        Pull(&obj2);
        Add(&obj1,&obj2,&res);
        Push(&res);
      pc=pc+1;
      break;
case SUB: //вычитание
      Pull(&obj1);
        Pull(&obj2);
        Sub(&obj1,&obj2,&res);
        Push(&res);
      pc=pc+1;
      break;
case MULT: //умножение
      Pull(&obj1);
        Pull(&obj2);
        Mult(&obj1,&obj2,&res);
        Push(&res);
      pc=pc+1;
      break;
case DIV: //деление
      Pull(&obj2);
        Pull(&obj1);
        Div(&obj1,&obj2,&res);     //res=ob1/ob2;
        Push(&res);
      pc=pc+1;
      break;
case JUMP://безусловный переход
       pc=prog[pc+1];
       break;
case EQU: //проверка на равно
       Pull(&obj1);
       Pull(&obj2);
       res.type=INT; //
       res.m.int_number=Equ(&obj1,&obj2);
       Push(&res);
       pc=pc+2;
       break;
case GRT: //проверка a>b
       Pull(&obj2);
```

```
        Pull(&obj1);
        res.type=INT; //
        res.m.int_number=Grt(&obj1,&obj2);
        Push(&res);
        pc++;
        break;
case LST: //проверка a<b
        Pull(&obj2);
        Pull(&obj1);
        res.type=INT; //
        res.m.int_number=Lst(&obj1,&obj2);
        Push(&res);
        pc++;
        break;
case BEQ: //условный переход  если истина
        Addr=prog[pc+1];
        Pull(&res);
        if(res.m.int_number) pc=Addr;
       else pc=pc+2;
        break;
case BNE: //условный переход  если ложь
        Addr=prog[pc+1];
        Pull(&res);
        if(!res.m.int_number) pc=Addr;
       else pc=pc+2;
        break;
case INC: //Инкримент
        Addr=prog[pc+1];
          Mem[Addr].m.int_number++;
          pc=pc+2;
          break;
case DEC: //Декримент
        Addr=prog[pc+1];
          Mem[Addr].m.int_number++;
          pc=pc+2;
          break;
case MOV: //переслать данные
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        Mem[Addr]=Mem[Addr2];
        pc=pc+3;
```

```
        break;
  case ILOAD: //косвенная загрузка данных
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        Addr3=prog[pc+3];
          Mem[Addr]=Mem[Addr2+Mem[Addr3].m.int_number];
        pc=pc+4;
        break;
  case ISAVE: //косвенное сохрание данных
        Addr=prog[pc+1];
        Addr2=prog[pc+2];
        Addr3=prog[pc+3];
          Mem[Addr2+Mem[Addr3].m.int_number]=Mem[Addr];
        pc=pc+4;
        break;
  case STOP: return 1;
  default: printf("Error command\n"); return -1;
  }
 }
}

#define X  0
#define Y  1
#define Z  2
unsigned char prog[]={
   IN,    X,
   IN,    Y,
   PUSH,  Y,
   PUSH,  X,
   DIV,         //x/y
   PULL,  Z,
   OUT,   Z,
   STOP
};

/*void main(){
 StackVirtualMachine(prog,Mem);        //выполение первой программы
 //SetMemory();
 //StackVirtualMachine(prog2,Mem);     //выполение второй программы
 getch();
} */
```

9.7.4 Транслятор для стековой виртуальной машины

```
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include <conio.h>
#include "stack_vm.h"

extern Object Mem[];  //внешняя ссылка на пямять по объекты

#define SIZEPROGRAM  500   //размер программы
#define SIZETABLVAR  30    //размер таблицы символов

struct {                  //описание таблицы символов
  int length;              //длина имени
  char Name[10];           //имя переменной
  } TablVar[SIZETABLVAR];
int TopVar;               //номер свобоной ячейки таблицы
char *ch;                 //указатель для разбора вводимой строки
char in_line[800];       //вводимая строка (буфер для хранения исходного текста)


void SkipBlanks();
void ExpressAdd(void);
void ExpressTest();
void Statement();

unsigned char program[SIZEPROGRAM]; //буфер для хранения выходного кода программы
int CurPos;                        //текщая позиция для записи кода

void OutCode(unsigned char Code){    //записать байта в выходной код программы
 if(CurPos<SIZEPROGRAM) program[CurPos++]=Code;
}
```

```
//
int GetCurrentPosition() { return CurPos; } //взять текущую позицию
void SetBrench(int pos, int addr) {       //установть значение операнда
 program[pos]=addr;
 }
//Печать ошибки
void Error(char *soo){
 printf("%s\n",soo);
}

// функция поиска имени в таблице символов
int FindNameVar(char *var, int *num){
  int i;
  int len;
  len=strlen(var);
  for(i=0; i<SIZETABLVAR; i++){
      *num=i;
      if(TablVar[i].length==0) return -1;
       if(TablVar[i].length==len)
       if(strcmp(TablVar[i].Name,var)==0) return i;
   }
   return -1;
}

// функция добавления нового имени в таблицу символов
int AddNameVar(char *var){
  int num;
  int len;
  char *ptr;
  if(FindNameVar(var,&num)==-1){
    if(TopVar<SIZETABLVAR){
      num=TopVar;
      len=strlen(var);
      TablVar[TopVar].length=len;
      strncpy(TablVar[TopVar].Name,var,9);
      TopVar++;
     }
     else Error("Переполнена таблица символов");
   }
  return num;
}
```

```
//функция взять имя во входной строке
int GetName(char *name, int max){   //здесь используются функции ctype.h
  int i;
  SkipBlanks();                    //пропустить пробелы
  if(isalpha(*ch)) *name++=*ch++;   //первый символ должен быть буквой
  else { *name=0; return 0;}
  for(i=1; i<max; i++){    //взять остальные символы (буква или цифра)
   if(isalnum(*ch))  *name++=*ch++;
   else {
     *name=0; return 1;             //имя нормальное
    }
  }
  *name=0;
  Error("Большое имя");
  return 0;
}

// функция пропуска  служебных символов
void SkipBlanks(){
 while(1){
  switch (*ch) {
   case 8:      //табуляция
   case 26:     //конец файла
   case 10:     //возврат каретки
   case 13:     //перевод строки
   case ' ':    //пробел
              ch++; break;
   default:
     return;
  }
 }
}

// проверка вхождения строки str во входной строке
int match(char *str){
  char *chv;
  SkipBlanks();      //пропустить служебные символы и пробелы
  chv=ch;
  while(*str)  {
   if((*str++)!=(*chv++)) return 0; //если не совпала строка то ноль
```

```
  }
  ch=chv;    //если совпала, то двигаем указатель
  return 1;
}

// проверка вхождения строки без сдвига указателя ch
int wait(char *str){
  char *chv;
  SkipBlanks();
  chv=ch;
  while(*str){
   if((*str++)!=(*chv++)) return 0;
  }
  return 1;
}

//функция проверки конца строки
int eof(){
 SkipBlanks();
 if(*ch=='\0') return 1;
 else return 0;
}

//функция пропуска одного символа
void skipchar() { if(*ch!='\0') ch++; }

// функция взятия числа во входной строки и преобразования к double
double getfloat(char *num, int max){ //используются функции <ctype.h>
 int i=0, j;
 SkipBlanks();          //пропустить пробелы
 if(*ch=='-'|| *ch=='+') //взять знак
              num[i++]=*ch++;
 SkipBlanks();          //пропусить пробелы
 for( ; i<max; i++){     //взять целую часть числа
  if(isdigit(*ch)) num[i]=*ch++;
  else break;
 }
 j=i;
 if(*ch=='.'){           //взять дробную часть числа
   num[i]=*ch++;
   for(j=i+1; j<max; j++)
```

```
   {
    if(isdigit(*ch)) num[j]=*ch++;
     else break;
   }
  }
 num[j]=0;
 return atof(num);
}

// функция разбора первичного выражения
void ExpressFunc(void){
 int num;
 char name[30];
 double f;
 int max=29;
 if(match("(")){  //вычисление выражения в скобках
    ExpressTest();
  if(match(")")==0) Error("Missmatch )");
  }
 else {         //вычисление функций
   //if(match("sin")) TranSin();
  //else
  // if(match("cos")) TranCos();
  //else
  // if(match("atan")) TranAtan();
  //else
  // if(match("log")) TranLog();
  //else
  // if(match("abs")) TranAbs();
  //else
   {
    if(GetName(name,max)==1){   //взятие значения переменной
      num=AddNameVar(name);    //найти  переменную в таблице
      OutCode(PUSH);          //сгенерировать команду Push <Addr>
      OutCode(num);
      }
    else  {                   //взять значение константы
      f=getfloat(name,max);
      if(name[0]=='\0') { skipchar(); return; }
      num=AddNameVar(name);    //найти или записать константу
      Mem[num].type=DOUBLE;    //занести значение в память
```

```
      Mem[num].m.float_number=f;
      OutCode(PUSH);          //сгенерировать команду Push <Addr>
      OutCode(num);
     }
    }
   }
}

// Функция  разбора выражение для  умножения и деления
void ExpressMult(void){
 ExpressFunc();       //вычислить первичное
 while(1) {
  if(match("*")) {     //если знак умножения
    ExpressFunc();    //вычислить второе первичное
    OutCode(MULT);    //сгенерировать  команду умножить
   }
 else
  if(match("/")) {    //если знак деления
    ExpressFunc();  //вычислить второе первичное
    OutCode(DIV);   //сгенерировать команду разделить
   }
  else break;
  }
}

// функция разбора выражения для сложения и вычитания
void ExpressAdd(void){
 ExpressMult();       //вычислить значение первого операнда
 while(1) {
  if(match("+")){      //если знак плюс
    ExpressMult();    //вычислить значение для второго опреанда
    OutCode(ADD);
   }
 else
  if(match("-")) {     //если это минус
    ExpressMult();    //взять значение второго операнда
    OutCode(SUB);
   }
  else break;
 }
}
```

```
// функция разбора логических выражений
void ExpressTest(void){
 ExpressAdd();       //вычислить значение первого операнда
 while(1){
  if(match(">")) {   //если знак больше
    ExpressAdd();    //вычислить значение для второго опреанда
    OutCode(GRT);
   }
 else
  if(match("<")) {   //если это меньше
    ExpressAdd();    //взять значение второго операнда
    OutCode(LST);
   }
  else
   if(match("=")){    //если равно
      ExpressAdd();   //взять значение второго операнда
      OutCode(EQU);
    }
  else break;
 }
}

// функция разьора оператора присваивания
void AssignOp(void){
 char name[10];
 int max=9;
 int num;
 if(GetName(name,max)==1)   //взять имя переменной
     {
      num=AddNameVar(name); //записать имя в таблицу
     }
     else  {
       printf("uncorect symbol %c\n",*ch);
       skipchar();
       return;
      }
   if(match(":=")) {         //имеется операция присваивания
      ExpressTest();         //разбрать выражение
      OutCode(PULL);         //сгнерировать команду PULL
      OutCode(num);
```

```
    }
}

//функция разбора условного оператора  if<>then<>else<>endif
void trans_if() {
  int pos_else;
  int pos_end;
  ExpressTest(); //разбор условия в стеке результат сравнения
  OutCode(BNE);  //если ложь переход на else или на endif
  pos_else=GetCurrentPosition(); //запомнить позицию для будущего заполе-
ния
  OutCode(0); //выделить место под будущее значение
  if(match("then")) {
    do {      //разбор операторов в блоке then
     Statement();
     } while(!wait("else")&&!wait("endif")&&!eof()); //ждать пока
    OutCode(JUMP);   //переход на конец цикла
    pos_end=GetCurrentPosition(); //запомнить позицию для будущего заполе-
ния
    OutCode(0); //выделить место под будущее значение адреса перехода
  }
  else Error("absent then");  //сообщение об ошибке
  if(match("else")){          //разбор блока else
    SetBrench(pos_else,GetCurrentPosition()); //установить адрес перехода на
else блок
    do { //разбор операторов else блока
     Statement();
     } while(!wait("endif")&&!eof());  //ждать конца endif или буфера
    SetBrench(pos_end,GetCurrentPosition());  //устанвить переход на конец
оператора if
  }
  else //если else блока нет
    SetBrench(pos_else,GetCurrentPosition()); //устанвить переход на конец
оператора if
  if(match("endif")) {
  }
}

//функция разбора операторы цикла  cycle<условие>  <тело> endcycle
void trans_while(){
 int pos_beg=GetCurrentPosition();
```

```
 int pos_end;
 ExpressTest(); //разбор условия, в стеке результат сравнения
 OutCode(BNE);  //если ложь переход
 pos_end=GetCurrentPosition();
 OutCode(0); //заполнить
 do {
  Statement();  //разбор операторов тела цикла
  }
 while(!wait("endcycle")&&!eof());  //ожидаем конца цикла
 if(match("endcycle")){
   OutCode(JUMP);   //переход в начало цикла
   OutCode(pos_beg);
   SetBrench(pos_end,GetCurrentPosition()); //выход за цикл
 }
 else Error("Missmatch endcycle");
}

//функция рабора оператора ввода  input <список переменных>
void trans_input(){
 char name[20];
 int max=19, num;
 do{
  if(GetName(name,max)==1)   //взятие значения переменной
      {
       num=AddNameVar(name);    //найти  переменную в таблице
       OutCode(IN);
       OutCode(num);
       }
 } while(match(","));   //продолжить если есть запятая
}

//функция разбора оператора вывода
void trans_output(){       //output <список переменных>
 char name[20];
 int max=19, num;
 do{
  if(GetName(name,max)==1)   //взятие значения переменной
      {
       num=AddNameVar(name);    //найти  переменную в таблице
       OutCode(OUT);
       OutCode(num);
```

```
      }
 } while(match(","));  //продолжить если есть запятая
}

//функция разбора одного из операторов
void Statement(){
 //char *beg=ch;
 if(match("cycle")) trans_while();   //разбор оператора цикла
 else
 if(match("if")) trans_if();        //разбор условного оператора
 else
  if(match("input")) trans_input();  //разбор оператора ввода
 else
  if(match("output")) trans_output();  //разбор оператора вывода
 else
  if(match(";")); //пустой оператор
 else
  AssignOp();     //разбор оператора присваивания
  //for(;beg!=ch; beg++) printf("%c",*beg); printf("\n");
}

//функция пропуска комментариев
void SkipCommentary(){
 while(match("--")) while(*ch!='\n'&&!eof()) skipchar();
}

//функция разбора текста программы вместе с крмментариями
void trans_program(){
 SkipCommentary(); //пропуск комментариев в начале программы
 if(match("program")){
 do{
  SkipCommentary(); //пропуск коментариев в теле программы
  Statement();
   } while((!wait("endprog"))&&(!eof()));
  }
  if(match("endprog")) OutCode(STOP);  //генерация команды останова
  else Error("Missmatch endprog\n");
  SkipCommentary(); //пропуск комментариев в конце программы
 }

// Функция вывода таблицы символов
```

```
PrintTablVar() {
 int i;
 for(i=0; i<TopVar; i++)  {
   printf("%2d %10s\n",i,TablVar[i].Name);
  }
}

//функция дезассемлирования программного кода виртуальной машины
void Disassembler(unsigned char *program){
 int next,pc=0;
 printf("Disassembler for stack vm (ver 1.00)\n");
 printf("-------------------------\n");
 do {
  printf("%03d  ",pc); //вывод адреса команды
  next=PrintCommand(&program[pc]);  //вывод команды
  pc+=next;
  } while(next!=0&&next!=-1);
 printf("--------- end  -----------\n");
}

//примеры программ
char example1[]= "program input x; x:=10/x; output x; endprog";

//программа вычисления суммы=(1+2+3+4+5)
char example2[]= "\
--this programm count sum \n\
--15.01.04\n\
 program\
 i:=1; s:=0;\
 cycle(i<5)\
  s:=s+i; i:=i+1;\
  output i;\
 endcycle\
 output s;\
endprog";

//программа вычисления факториала
char example3[]= "program\
 --factorial(10)\n\
 i:=1; s:=1;\
 cycle(i<10)\
```

```
  s:=s*i; i:=i+1;\
  output i;\
 endcycle\
 output s;\
endprog";


main(){ //

  TopVar=0;           //инициализация стека и таблицы переменных
  //printf("trans>");   //вывод подсказки
  //gets(in_line);      //ввод строки символов
  //strcpy(in_line,"program x:=10/x if(c>20) then c:=5 else c:=6 endif endprog");
  //strcpy(in_line,"program x:=10/x cycle(c>20) c:=c+1 endcycle endprog");
  strcpy(in_line,example2);
  ch=&in_line[0];     //инициализация указателя разбора
  printf("%s\n",in_line);
  trans_program();
  Disassembler(program);

  PrintTablVar();
  StackVirtualMachine(program,Mem);
  getch();
}
```

ЛИТЕРАТУРА

1. Ахо А., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструменты: Пер. с англ. — М.: Издательский дом «Вильямс», 2001. — 768 с.
2. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. — М.: Мир, 1979.

3. Ахо А., Ульман Дж.Д. Теория синтаксического анализа, перевода и компиляции. — М.: Мир, 1978. — Т.1. — 616 с.; Т.2. — 488 с.

4. Ахо А., Ульман Дж.Д. Принципы машинного проектирования. — М.: Мир, 1983. — 352 с.

5. Грис Д. Конструирование компиляторов для цифровых вычислительных машин. — М.: Мир, 1975. — 544 с.

6. Рейуорт-Смит В.Дж. Теория формальных языков. Вводный курс. М.: Радио и связь, 1988. — 128с.

7. Вайнгартен Ф. Трансляция языков программирования. — М.: Мир, 1977. — 192 с.

8. Касьянов В.И., Поттосин И.В. Методы построения трансляторов. — Новосибирск: Наука, 1986. — 344 с.

9. Льюис Ф., Розенкрац Д., Стирнз Р. Теоретические основы проектирования компиляторов. — М.: Мир, 1979. — 656с .

10. Хантер Р. Проектирование и конструирование компиляторов. — М.: Финансы и статистика, 1984. — 232 с.

11. Зелковиц М., Шоу А., Бынон Дж. Принципы разработки программного обепечения. — М.: Мир, 1982.

12. Бекхауз Р.Ч. Синтаксис языков программирования. — М.:Мир, 1986. — 281 с.

13. Бек Л. Введение в системное программирование. — М.: Мир, 1988.

14. Хэндрикс Д. Компилятор языка Си для микро-ЭВМ. — М.: Радио и Связь, 1989. — 239 с.

15. Берри Р. Микинз Б. Язык Си: введение для программистов — М.: Финансы и статистика, 1988. — 191 с.