6. Полная программа на Паскале

Программист – это создатель вселенных, для которых он является единственным законодателем. Ни один драматург, ни режиссер, ни император в истории никогда не обладал такой абсолютной властью при постановке спектакля или на поле битвы и не командовал такими непоколебимо верными делу актерами или воинами.

Джозеф Вейценбаум,
автор программы «Элиза», 1966 г.

Теперь пришло время написать полную программу и обсудить её семантику. Рассмотрим совсем простую задачу. Пусть необходимо ввести одно целое число в переменную `x` оператором `read(x)`, затем присвоить переменной `y` значение `x+1` и вывести полученное значение `y`. На Паскале для этого обычно пишут такую программу.

```
program A(input,output);
  var x,y: integer;
begin
  read(x); y:=x+1; Write(y)
end.
```

А теперь зададимся естественным вопросом, когда эта программа будет давать правильный ответ? Ясно, что для этого, необходимо наложить определённые ограничения на её входные данные. Эти ограничения можно записать в виде некоторого логического утверждения, зависящего от входных данных, в теории программирования это называется предусловием (precondition) программы. Какое же предусловие будет в нашем случае?

Во-первых, необходимо потребовать, чтобы в стандартном входном потоке данных `input`, к которому обращается оператор `read(x)`, находилась правильная лексема целого числа. Далее, после преобразования её во внутреннее представление, это целое число должно принадлежать диапазону представимых в нашей Паскаль-машине целых чисел. Обозначим этот диапазон буквой `Z` (по аналогии с обозначением в математике), тогда это оба эти требования можно записать как `x∈Z`. Далее, надо потребовать, чтобы величина `x` была меньше, чем самое большое представимое в Паскале целое число, которое, как известно, имеет стандартное имя `MaxInt`. Теперь можно написать предусловие нашей программы в виде:

`Pred={x∈Z ∧ x<MaxInt}` [1]

Будем здесь, как в математических текстах, для краткости знак ∧ использовать для операции Паскаля **`and`**, а знак ∨ – для операции **`or`**. При истинности предусловия программа должна выдать правильный ответ, что можно записать в виде логического постусловия (postcondinion) программы:

`Post={y∈Z ∧ y=x+1}`

Дадим теперь определение, когда (не имеющая синтаксических ошибок) программа является (семантически) *правильной*. Сначала поймём, что правильность или неправильность программы зависит от её предусловия. Например, приведённая выше программа для предусловия `Pred={x∈Z}` будет *неправильной*, так как для введённого `x=MaxInt` правильного ответа не получиться, т.е. постусловие не будет выполняться. Предусловие, сама программа и постусловие носят в теории программирования название триада Хоара.

---

[1] Более точно оно называется *слабейшим* (weakest) предусловием. Действительно, можно взять и «более грубое» предусловие, например, `Pred={x∈Z ∧ x<0}`, в нём постусловие, конечно, тоже будет выполняться. Слабейшее предусловие обеспечивает самое большое множество допустимых входных данных [12].

Здесь используются элементы так называемой аксиоматической семантики (логики) Т. Хоара.[1] Отметим, что можно рассматривать применение предусловий и постусловий не только ко всей программе целиком, но и к составляющим её операторам, но мы этого делать не будем. Более подробно о предусловиях и постусловиях можно почитать в книгах [13,14]. Мы познакомимся только с самыми основами аксиоматической семантики. Ранее, в разд. 2.2 мы немного говорили об операционной семантике, когда рассказывали о венском методе описания формальных языков.

Итак, будем называть программу правильной в некотором предусловии, если при истинности этого предусловия программа завершится и обязательно обеспечит истинность нужного нам постусловия.[2] Здесь можно провести аналогию с работой завода: при соответствии входного сырья заданным стандартам, *правильно* работающий завод должен обеспечить выпуск продукции, тоже удовлетворяющей заданным стандартам, иначе завод работает *неправильно*.

В том случае, если предусловие не выполнено, программа, вообще говоря, не обязана обеспечить правильность постусловия. Другими словами, программа при этом может зациклиться, аварийно остановиться (ошибка времени выполнения – Run Time Error, по-русски АВОСТ), или же выдать неверный результат. По аналогии с заводом, при получении им плохого сырья он не обязан производить хорошую продукцию, а может выдать брак или повести себя и совсем нехорошо. Например, металлургический завод может взорваться, если в привезённом на переплавку металлоломе окажется старый неразорвавшийся снаряд.

Как Вы догадываетесь, такое неправильное поведение программы не обрадует заказчика, который поручил программисту написать эту программу. Исходя из этого, с точки зрения нашего заказчика, лучшим для нашей программы было бы предусловие

```
Pred=true
```

Такое предусловие следует известному принципу хорошего обслуживания «Клиент всегда прав». В этом предусловии написанная выше программа, конечно же, будет *неправильной*. Разберёмся сначала, а какое постусловие должно быть у нашей программы в таком, тождественно истинном, предусловии. Естественно, оно тоже должно быть тождественно истинным, т.к. все входные данные по определению «хорошие». Можно, например, для этого случая предложить следующее постусловие

```
Post={x∈Z ∧ x<MaxInt ∧ y∈Z ∧ y=x+1 ∨
      x∉Z ∧ 'Плохое x' ∨
      x∈Z ∧ x=MaxInt ∧ 'Большое x'}
```

Как Вы можете убедиться, данное постусловие (логическая сумма трех логических слагаемых) является тождественно истинным, так как для любого значения, введённого из входного потока, одно из трёх логических слагаемых обязательно будет истинным.

В математической логике тождественно истинные формулы называются *общезначимыми* или *тавтологиями*.

Как можно догадаться, написать программу для решения задачи в тождественно истинном предусловии будет совсем не просто. В то же время, очевидно, что «настоящие» программы должны быть написаны именно в таком предусловии (как уже говорилось, «Клиент всегда прав»). У хороших программ не должно быть зацикливаний, выдачи непонятных диагностик и «синего экрана смерти». Позже мы напишем программу для решения этой задачи в тождественно истинном предусловии.

В качестве следующего примера рассмотрим такую простую программу (заголовок программы со словом **program** больше для краткости писать не будем).

```
  var a,b,c: real;
begin
  read(a,b); c:=sqrt(sqr(a)+sqr(b)); Write(c)
end.
```

---

[1] Сэр Чарльз Энтони Ричард Хóар (Charles Antony Richard Hóare) английский учёный, специалист в области надёжности программного обеспечения и взимодействия параллельный вычислительных процессов. Автор знаменитого алгоритма быстрой сортировки. В 1959-60 годах учился в МГУ у академика А.Н.Колмогорова ⚠.

[2] В научной литературе в этом случае говорят о *полной* правильности программы. *Частичная* правильность подразумевает, что если программа завершится, то постусловие будет выполнено.

Можно предположить, что программа решает следующую задачу: ввести длины двух катетов a и b, затем вычислить и вывести длину c гипотенузы прямоугольного треугольника. Правильная ли эта программа? Теперь Вы уже должны понимать, что этот вопрос не так прост и имеет смысл только для *конкретного предусловия* данной задачи. Синтаксически верная программа может быть правильной в одном предусловии и неправильной в другом. Для написания предусловия, в котором приведенная выше программа будет правильной, сначала, естественно, придётся осознать поставленную задачу и выполнить её спецификацию.

Ясно, что для введённых *отрицательных* значений длин катетов естественно предположить, что треугольник не существует и программа должна давать соответствующую диагностику. Сложнее обстоит дело с нулевыми значениями длин катетов. В классической математике такие треугольники, конечно, не существуют (вырождаются в отрезок или точку), но Вы уже должны знать, что компьютерные программы «живут» в мире особой, *дискретной* математики. В частности, вводимые из «внешнего мира» вещественные числа округляются до ближайшей точки на оси вещественных чисел Паскаль машины. При этом введённая ненулевая лексема вещественного числа, после преобразования её во внутреннее представление вполне может округлиться до вещественного машинного нуля. Исходя из этого, лучше будет считать (это спецификация), что у таких *вырожденных* треугольников гипотенуза существует и вычисляется по той же самой формуле, т.е. будет совпадать с другим ненулевым катетом или вообще обращаться в ноль.

Напишем предусловие для задачи вычисления длины гипотенузы. По аналогии с классической математикой, обозначим через R конечное множество всех представимых вещественных чисел на оси Паскаль-машины, а внутренние представления длин катетов a и b и гипотенузы c обозначим как $\underline{a}$, $\underline{b}$ и $\underline{c}$. Тогда можно написать такое хорошее для заказчика предусловие задачи:

$$Pred=\{\underline{a},\underline{b}\in R \wedge \underline{a},\underline{b}\geq 0 \wedge \underline{c}=sqrt(\underline{a}^2+\underline{b}^2)\in R\}$$

с таким постусловием

$$Post=\{\underline{a},\underline{b}\geq 0 \wedge \underline{c}=sqrt(\underline{a}^2+\underline{b}^2)\in R \vee (\underline{a}<0 \vee \underline{b}<0 \vee sqrt(\underline{a}^2+\underline{b}^2)\notin R) \wedge \text{'Плохие данные'}\}$$

Другими словами, предполагается, что, если для введённых значений катетов гипотенуза представима (существует) на вещественной оси Паскаль-машины, то она должна быть вычислена и выведена программой. В этом, естественном для заказчика, предусловии написанная выше программа *неправильная*! Действительно, даже если длина гипотенузы $\boxed{\underline{c}\in R}$, это не значит, что она будет получена программой, т.к. значения $\underline{a}^2$, $\underline{b}^2$ или $\boxed{\underline{a}^2+\underline{b}^2}$ могут не принадлежать R. Предусловие для написанной выше программы, в котором она правильная, следующее

$$Pred=\{\underline{a},\underline{b}\in R \wedge \underline{a},\underline{b}\geq 0 \wedge \underline{a}^2+\underline{b}^2\in R\}$$

Это может не удовлетворить заказчика, т.к. многие *представимые* в Паскале значения гипотенуз для *представимых* значений катетов не будут получены программой, которая при этом будет аварийно завершаться с не очень понятной для пользователя диагностикой типа Real Overflow. В то же время следует понять, что можно написать правильную программу и для «хорошего» предусловия. Это, например, такая программа:

```
  var a,b,c,max: real;
begin
  read(a,b);
  if (a>=0.0) and (b>=0.0) then begin
    if a>b then max:=a else max:=b;
    if max=0.0
       then c:=0.0
       else c:=max*sqrt(sqr(a/max)+sqr(b/max));
    Write(c)
  end else writeln('Плохие данные')
end.
```

В этой программе из-под квадратного корня вынесена длина бóльшего катета, так что величина подкоренного выражения будет не больше двух. Таким образом, если гипотенуза представима в этом типе данных, то она будет этой программой получена. Разумеется, новая программа существенно

сложнее старой версии. Такие программы, которые всегда пытаются получить ответ, если он представим, называются в программистской литературе робáстными (robust). К сожалению, хорошего русского термина нет, буквально это программы «крепкие», в смысле устойчивые к «плохим» входным данным. Как Вы догадываетесь, хорошо написанные программы (программный продукт) должны быть робастными, хотя это и не так просто сделать.[i] [см. сноску в конце главы]

Рассмотренные примеры призваны показать, с какой осторожностью программист должен относится к «очевидным» решениям даже самых простых задач.

Большинство современных языков программирования не позволяют записывать в программе предусловие и постусловие иначе, чем в виде комментария. В некоторых языках, однако, можно использовать специальные логические утверждения, истинность которых должна проверяться во время счёта программы. Одним из первых это было реализовано в языке Эйфель [15], в котором предусловие и постусловие являются *директивами* для соответствующего исполнителя и могут учитываться при выполнении программы (с двух `--` в этом языке начинается комментарий):

```
require x<Maxint   -- Предусловие
ensure y=x+1       -- Постусловие
```

Бертран Мейер, автор языка Эйфель, предложил для такой разработки программного обеспечения специальный термин контрактное программирование, что хорошо отражает суть дела. Из достаточно широко используемых языков можно назвать наш Free Pascal и Python [16] с директивами-утверждениями **assert**. Такие же средства предусмотрены и в новом стандарте языка C++ 20:

```
[[ expect  x<Maxint ]]  // Предусловие
[[ ensures y=x+1 ]]     // Постусловие
```

Отметим, что идея контрактного программирования появилась ещё в первом алгоритмическом языке высокого уровня Планкалкэль (Plankalkül – Исчисление планов), созданном немецким инженером-конструктором ЭВМ К. Цузе в 1946 году. В этом языке уже были стандартные типы данных, массивы, цикл с постусловием и даже обработка исключений, в вот оператора **goto** не было ⚠.

Рассмотрим теперь пример, который часто встречается в книгах по программированию. Требуется обменять значения двух переменных без использования третьей (вспомогательной) переменной. Обычно предполагается следующее решение этой задачи (для целых и вещественных чисел):

```
var x,y: integer; a,b: real;
    y := y-x; x := x+y; y := x-y;
    b := b-a; a := a+b; b := a-b;
```

При проведении этой операции в *физическом мире* сразу возникают проблемы. Например, предложение обменять содержимое двух вагонов, в одном из которых песок, а в другом уголь, причём без использования третьего вагона, обычно сразу отвергается, как невыполнимое. Немного подумав, можно, однако, иногда предложить решение этой задачи. Например, если песок занимает не весь вагон, то можно сгрести его в одну сторону и установить в вагоне перегородку. После этого уголь из второго вагона пересыпать в свободную часть вагона с песком, а песок затем перегрузить в освободившийся от угля вагон. Теперь осталось только убрать ненужную больше перегородку.

Так можно ли обменять значения двух переменных без использования третьей? Поразмыслив над таким решением, можно сообразить, что в качестве предусловия этой задачи необходимо потребовать, чтобы сумма значений переменных принадлежали множествам Z и R. Другими словами, только в половине случаев предложенное выше решение будет давать правильный ответ!

Как уже упоминалось выше, на большинстве ЭВМ реализована целочисленная арифметика с дополнительный кодом, поэтому результат такого обмена для целых переменных будет правильным. А вот для вещественных переменных, как и в физическом мире, в половине случаев ответ будет неверным.

В то же время, во многих языках программирования высокого уровня реализованы особые битовые операции над целочисленными аргументами, в частности, операция **xor** (неэквивалентность, сложение по модулю два). С помощью этой операции также можно обменять значения двух переменных без использования третьей переменной. Например, в языке Free Pascal:

```
var x,y: longint; a: single absolute x;
    b: single absolute x;
```

```
y:=y xor x; x:=x xor y; y:=x xor y;
```

Как видим, для обмена вещественных переменных мы просто совмещаем их в памяти с целочисленными переменными. Можно задуматься над вопросом, как это соотносится с приведенной выше операцией обмена содержимым двух вагонов. [ii] [см. сноску в конце главы]

Приведённые примеры должны помочь Вам проводить правильную спецификацию задачи. При решении задач на самостоятельных и контрольных работах, зачетах и экзаменах, вполне можно позволить себе «щадящее» предусловие вида «все входные данные хорошие, иначе пусть будет неправильный ответ или стандартная аварийная диагностика». При решении же задачи на практикуме на ЭВМ этого уже делать не стоит, спецификация должна быть согласована с преподавателем (он считается заказчиком программы 🙂).

6.1. Примеры решения задач

…будучи молодым и наивным программистом, я смотрел в текст своей программы и видел в нём только то, что хотел видеть, а не то, что там написано в реальности.

Dukarav, программист

Рассмотрим несколько типичных простых задач и трудности, которые могут встретиться при их решении. В первой задачи требуется ввести любое целое число `x` оператором ввода `read(x)` и вывести сумму значений всех десятичных цифр этого числа. Таким образом, надо решить задачу в предусловии `Pred={x∈Z}`. Осознание задачи, должно получиться:

`1234 ⇒ 10` `0 ⇒ 0` `-5067 ⇒ 18`

Ясно, что надо в цикле найти все цифры введённого числа и просуммировать их. Максимальное число цифр неизвестно (зависит от реализациии типа целого числа). Так как в каждом числе не менее одной цифры, то лучше использовать цикл с постусловием, тело которого выполняется не менее одного раза. Обычно учащиеся предлагают такое решение:

```
  var x,S: integer;
begin read(x); S:=0; { сумма цифр }
  while x<>0 do begin
    S:=S+x mod 10; { прибавить последнюю цифру }
    x:=x div 10    { убрать последнюю цифру }
  end;
  writeln(S)
end.
```

Здесь допущена серьезная ошибка, так как операция `x mod 10` для *отрицательного* числа даёт *отрицательное* значение цифры, так что будет `-5067 ⇒ -18`. Пытаясь исправить эту ошибку, учащиеся обычно после ввода числа добавляют в программу оператор `x:=abs(x)`. Одна ошибка исправляется, но вносится другая, чтобы её понять, рассмотрим предусловие исправленной программы:

`Pred={x∈Z ∧ ∃ abs(x)}`

А теперь надо вспомнить, что целочисленная ось Паскаля несимметрична, отрицательных чисел на единицу больше, чем положительных. Таким образом, существует целое число, у которого нет абсолютной величины, значит для одного из введённых чисел наша программа не даст правильного ответа. Для математика это можно сравнить, например, с потерей одного корня при решении уравнения. Обычно программисты не обращают на это внимание, однако для важных программ (например, управляющих химическим реактором) это может «неожиданно» создать серьёзную неприятность.

Можно, например так решить эту проблему:

```
  var x,S: integer;
begin read(x); S:=0;
  while x<>0 do begin
    S:=S+abs(x mod 10);
    x:=x div 10
```

```
  var x,S: integer;
begin read(x); S:=0;
  while x<>0 do begin
    S:=S+x mod 10;
    x:=x div 10
```

```
  end;
  writeln(S)
end.
```

```
  end;
  writeln(abs(S))
end. { Это решение лучше }
```

Итак, в режиме {$R-} функция `abs(x)` может дать неверный результат, а в режиме {$R+} дать аварийный останов 😕. Как же безопасно вычислить абсолютную величину целого числа? Приходится действовать как-то так:

```
var x: integer;
begin {$R-}
  read(x); x:=abs(x);
{ abs(Minint)=Minint - архитектура ЭВМ! }
  if x<0
    then Write('У ',x,' нет модуля')
    else Write('Модуль ',x,'=',x)
end.
```

Рассмотрим теперь такую задачу. Из входного потока `input` надо ввести текст до точки (признака конца ввода). Программа выводит сумму значений всех цифр в таком тексте. Предусловием нашей программы будет существование символа точки во вводимом тексте (иначе пусть программа зациклится), и то, что сумма цифр в тексте не больше, чем `Maxint` (иначе пусть программа выдаст неправильный ответ). В качестве решения можно предложить такую программу

```
  var c: char; S: integer;
begin S:=0; { сумма цифр }
  repeat
    read(c); { c='0' не берём! }
    if ('1'<=c) and (c<='9') then
       S:=S+ord(c)-ord('0')
  until c='.';
  writeln(S)
end.
```

При решении мы воспользовались тем, что в любом алфавите символы цифр идут подряд, значит для цифры выполняется условие `'0'≤c≤'9'`. Далее ясно, что значением, скажем, цифры `'5'` служит её порядковый номер в алфавите, из которого вычтен порядковый номер нуля.

В программе осталась одна маленькая «хитрая» ошибка, попытайтесь её найти и исправить.

Последней рассмотрим такую задачу. Из входного потока `input` надо ввести текст до точки (признака конца ввода). Программа выводит процент строк этого текста, которые начинаются с символа звёздочки `'*'`. Так как в тексте не менее одной строки (в конце которой стоит точка), то задача всегда имеет решение. Будем выдавать ответ с двумя знаками после запятой, например, `12.34%`.

Здесь важно провести правильное осознание задачи. Ясно, что теперь мы уже не можем «бездумно» вводить символ за символом, так как после выдачи последнего символа *текущей* строки почти сразу же будет выдан первый символ *следующей* строки, и факт перехода с одной строки на другую мы не зафиксируем. Кроме того, строки текста могут быть пустые (не содержать ни одного символа), что тоже надо учитывать в алгоритме.

Здесь нам поможет стандартная функция Паскаля `eoln`, которая возвращает **`true`**, когда текущая строка в буфере ввода закончилась, и при очередном вводе символа в буфер поступит следующая строка. С помощью этой функции можно предложить такое решение нашей задачи:

```
  var c: char; N,Nz: integer;
begin N:=0; Nz:=0;
❶ c:='␣'; { на случай, если первая строка пустая }
  repeat
    if not eoln then begin { строка не пустая }
       read(c);
       if c='*' then Nz:=Nz+1;
       while not eoln do read(c)
```

```
        { в c последний символ строки }
    end;
    N:=N+1; ❷ readLn { переход на след. строку }
  until ❸ c='.';
  Writeln(Nz/N*100:6:2,'%')
end.
```

В точке ❶ для стандарта Паскаля значение переменной c ничему не равно, поэтому мы присвоили этой переменной конкретное значение пробела, иначе, если первая строка пустая, то в точке ❸ поведение программы не определено. Отметим, что для языка Free Pascal в точке ❶ значение `c=#0` и дополнительно ничего делать не надо.

Здесь Вам надо понять, что в предусловие задачи включены ограничения `N,Nz≤Maxint`, хотя решение задачи существует всегда. Попытайтесь изменить программу, чтобы убрать эти ограничения (предполагается, что мы математики 🙂). Далее, в нашей программе есть хитрая ошибка: если последняя строка текста имеет, например, вид

```
ABC.DEF
```

то наша программа зациклится 🐷. Видимо, мы не до конца осознали условие «вводить текст до точки», оно вовсе не предполагает, что эта точка стоит в конце последней строки! Попытайтесь исправить и эту ошибку.

Особое внимание обратите на оператор ❷ `readln`, он удаляет из буфера ввода все символы текущей строки, включая служебные символы, задающие признак конца сток. В разных операционных системах текстовые строки имеют разные признаки конца: `\n=LF=#10` в ОС семейств Unix и Android и два символа `CR`, `LF` (`#13`, `#10`) в ОС Windows. Паскаль, однако, является платформонезависимым языком, приведённая выше программа будет работать правильно везде. Попытка использовать при обработке строк на Паскале символы `\n`, `LF` и `CR` является ОШИБКОЙ.

Итак, даже самые простые примеры показывают, как внимательно надо относится к осознанию, спецификации и отладке программ.

6.2. Структура программы

Цель части – как одной, так и многих –
целесообразность всей структуры.
Клавдий Гален, 129-216 н.э.

Вернёмся к нашей первой программе прибавления к переменной единицы

```
  var x,y: integer;
begin
  read(x); y:=x+1; Write(y)
end.
```

Теперь пришло время реализовать этот алгоритм в виде «хорошей» программы с предусловием

```
Pred=true
```

и постусловием

```
Post={x∈Z ∧ x<MaxInt ∧ y∈Z ∧ y=x+1 ∨
      x∉Z ∧ 'Плохое x'             ∨
      x∈Z ∧ x=MaxInt ∧ 'Большое x'}
```

Сначала нам предстоит разобраться, какие функциональные части (компоненты) содержит «хорошая» программа. Это такие части:

1. Алгоритм, преобразующий (правильные) входные данные в результат работы.
2. Контроль правильности входных данных.
3. Диалоговый интерфейс.

Как мы уже знаем, реализация первых двух пунктов сильно зависит от предусловия и постусловия программы. Без второго пункта не обходится ни один программный продукт, например, для электронной таблицы в числовое поле ввести «не число» не получится. Третий пункт присутствует только в так называемых *диалоговых* программах, для которых предполагается во время счёта вести диалог между программой и пользователем.

Раньше программа могла вести диалог только с человеком, но сейчас одна программа или же человек может «разговаривать» и с чем-то другим. Как Вы знаете, в Интернете большой проблемой является определить, ведётся ли диалог с человеком, или с «искусственным интеллектом». Особенно любопытно, когда две такие системы (скажем, две Алисы на разных смартфонах) ведут диалог между собой.

Для «распознавания» человека могут применяться различные методы, например, капчи (CAPTCHA – Completely Automated Public Turing test to tell Computers and Humans Apart), это частный случай так называемого теста Тьюринга на разумность.[1] Сейчас ежедневно люди (и «продвинутые» программы) вводят более миллиарда таких подтверждений своей «человечности». Как видно, для роботов применяется дискриминация 🙂. Интересно, что сейчас каждый человек (и каждая программа) могут послать такую капчу на специальные сайты, где сидящие там за терминалами (восточные) люди (за «смешные» деньги) введут правильный ответ. Интересно, что на эти же сайты «ходят» и системы искусственного интеллекта.

Сначала рассмотрим проблему контроля входных данных. Эта проблема является актуальной во многих задачах, поэтому системы программирования (в нашем случае система программирования языка Free Pascal) предоставляет для этого некоторые стандартные средства, которые мы и рассмотрим. Логическое слагаемое постусловия

```
x∉Z ∧ 'Плохое x'
```

предполагает, что некорректная или выходящая за допустимые значения вводимая величина вызывает соответствующую диагностику. Чтобы понять, как это происходит, вспомним схему работы процедуры стандартного ввода `read(x)`. Сначала эта процедура читает из стандартного входного потока Паскаля `input` символ за символом, в попытке собрать из этих символов правильную лексему целого числа, эта лексема заканчивается символом пробела, табуляции или концом строки. При поступлении неправильной лексемы целого числа фиксируется ошибка времени выполнения `Input/Output Error` и выполнение программы завершается.

Ясно, что вместо диагностики `Input/Output Error` нам бы хотелось продолжить выполнение программы и выдать свою диагностику `'Плохое x'`. Для обеспечения этой и многих других возможностей по управлению вводом/выводом, в языке Free Pascal пользователю предоставляется набор новых стандартных процедур, функций, переменных и констант. Все они описаны в модуле с именем `Crt`, а чтобы сделать эти новые стандартные имена доступными, в начале программы следует поместить предложение `uses Crt;`

`Crt` (Cathode Ray Tube) это электронно-лучевая трубка, на основе которой строились первые мониторы и телевизоры. В *графическом* режиме лучше использовать модуль `WinCrt` (или аналогичные в других ОС). Учтите, что современные системы программирования для обработки аварийных ситуаций (и, в частности, `Input/Output Error`) используют более современный метод, так называемый перехват исключений. Вы можете посмотреть, как это делается, в главе 17). Мы же будем действовать «по простому».

Чтобы для плохой лексемы не выдавать фатальную диагностику `Input/Output Error`, где-то перед описанием процеды ввода `read(x)` следует записать директиву `{$I-}`. При этом в случае поступления плохой лексемы аварийная диагностика не выдаётся, а выполнение программы продолжается с оператора, следующего за оператором ввода. Переменная `x` при этом, естественно, не получает никакого (хорошего) значения. Таким образом, директива `{$I-}` при ошибке ввода/вывода *блокирует* прекращение счёта программы и выдачу стандартной аварийной диагностики.

Как же теперь узнать, прошла ли операция ввода/вывода нормально, или же нет? Для этого предусмотрена целочисленная стандартная функция без параметров с именем `IOResult`. Она выдаёт целое число, которое называется кодом возврата для последней операции ввода/вывода. Нулевое значение кода возврата означает успешное окончание последней операции ввода/вывода, а остальные значения свидетельствуют о каких-либо ошибках. Например, если поток `input` подключён к клавиатуре, то различные ненулевые значения кода возврата могут означать поступление плохой лексемы (код=106), неисправность клавиатуры, исчерпание (закрытие) входного потока и т.д. Функция

---

[1] Первоначальный вариант этого теста предложен А. Тьюрингом в 1950 году.

IOResult имеет одну особенность: выдача кода возврата происходит с его очисткой, так что спросить, произошла ли ошибка конкретного ввода, можно только один раз. При повторном чтении эта функция выдаст нулевой код возврата, поэтому лучше сразу записать этот код в какую-нибудь целочисленную переменную, например: `kv:=IOResult`.

«Настоящий» код возврата Free Pascal хранит в стандартной переменной с именем InOutRes. Использовать переменную InOutRes «напрямую», читая из ней код возврата, настоятельно не рекомендуется, её наличие связано с историческими причинами.

Учтите, что в режиме {$I-} при возникновении ошибки только чтение кода из IOResult (с его очисткой) делает возможным выполнение последующих операций ввода/вывода.

Далее, для правильной введённой лексемы процедура read пытается преобразовать эту лексему во внутреннее представление целого числа. Так как этот этап работает для ввода в целые переменные любого типа, то процедура сначала строит внутреннее представление целого числа самого большого диапазона. Это знаковый тип int64 и беззнаковый тип qword, их объединённый диапазон от -2^{63} до $+2^{64}-1$, при выходе введённого значение за этот диапазон тоже выдаётся ненулевой код возврата.

Затем полученное значение присваивается вводимой переменной, в нашем случае переменной x типа integer, причем при выходе введённого значения за диапазон типа integer, поведение процедуры read, как Вы уже знаете, зависит от значения параметра в директиве компилятора {$R±}. При отсутствии этой директивы, или её задания в виде {$R-} вводимой переменной присваивается *неправильное* (усечённое) значение, а при задании директивы в виде {$R+} происходит ошибка времени выполнения Range Error. Исходя из изложенного алгоритма работы процедуры read ясно, что сначала нужно вводить целое число не в переменную x типа integer, а во вспомогательную переменную temp типа int64, что мы и будем делать в нашей программе.

Рассмотрим теперь основные правила организации диалогового интерфейса с пользователем. Будем рассматривать простейший случай, когда ввод данных производится с клавиатуры, а вывод – на дисплей в текстовом режиме. Диалог ведётся между программой и пользователем (человеком), причём ведущей в этом диалоге является программа. Именно программа выдаёт приглашения к вводу данных (в том числе выбору из списка действий из меню), сообщения об ошибках и т.д.

Дисплей в *основном* текстовом режиме обладает следующими характеристиками. Он состоит из 25 строк, пронумерованных от 1 до 25 (по возрастанию координаты y), в каждой строке содержатся ровно 80 символов, пронумерованных от 1 до 80 (по возрастанию координаты x, см. рис. 6.1). Без крайней необходимости не рекомендуется переключать дисплей в другие текстовые режимы, содержащие большее или меньшее число строк и столбцов, так как текст становится плохо читаемым (особенно при работе не в полноэкранном, а в оконном режиме). Текущий размер экрана хранится в стандартных переменных модуля Crt с именами WindMaxX и WindMaxY. Присваивая этим переменным новые значения можно (в определённых пределах) увеличивать и уменьшать размер текущего экрана.

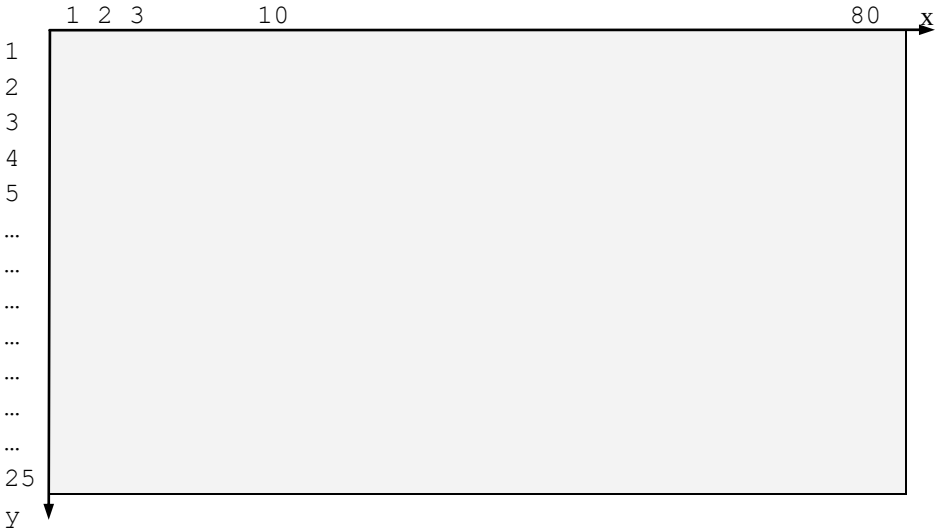

Рис. 6.1. Вид дисплея в стандартном текстовом режиме.

На пересечении строки и столбца находится знакоместо, в которое может быть выведен один символ. У каждого знакоместа есть несколько свойств или *атрибутов*, важнейшими из которых являются цвет символа и цвет фона, на котором изображается этот символ. Множество этих цветов образует цветовую палитру. Минимально возможное число цветов, естественно, два, обычно это светло-серые символы на чёрном фоне. Далее следуют палитры из 16, 256, 2^{16}, 2^{24} и 2^{32} различных цветов, бóльшее количество цветов не имеет смысла, так как человеческий глаз перестаёт их различать.

В качестве примера рассмотрим, как работать в палитре из 16 цветов, которые пронумерованы целыми числами от 0 до 15. Можно не заучивать, какой номер соответствует какому цвету, так как в модуле `Crt` все цвета описаны в виде имён стандартных констант:

```
const Black=0; Blue=1; Green=2; …
```

Проще всего вести набор текста программы в так нациваемом интегрированнои режиме, когда редактор текста на Паскале входит в состав системы программирования. При этом, чтобы увидеть описания констант цветов, достаточно набрать при редактировании текста программы имя одной из них, например, `Blue`, подвести курсор к этому имени и нажать клавиши `Ctrl-F1` (или `^F1`). Это соответствует запросу к информационной подсистеме помощи языка Free Pascal: «Что известно об этом имени?». Такие же вопросы можно задавать и об остальных именах языка Паскаль.[1]

В каждый момент времени определены *текущие* цвет фона и цвет символа, именно этими цветами будет выводить данные процедура `write`. Разумеется, в модуле `Crt` описаны стандартные процедуры для независимой смены цвета фона и цвета символа, но часто необходимо одновременно сменить *оба* эти цвета, для чего можно использовать такой приём. В модуле `Crt` описана целочисленная переменная со стандартным именем `TextAttr`, в которой, в частности, хранится информация о текущих цветах. Эта информация хранится в виде двух шестнадцатеричных цифр в формате:

`16*`<цвет фона>`+`<цвет букв>

Для одновременной смены этих цветов достаточно присвоить этой переменной новое значение, например

```
TextAttr:=16*Blue+Yellow
```

После этого вывод на экран будет производиться жёлтыми буквами на синем фоне. *Стирание* экрана, т.е. вывод во все его позиции символа пробела текущим цветом фона, производится вызовом стандартной процедуры модуля `Crt` с именем `ClrScr`. В текущей позиции экрана находится *курсор*, именно с этой позиции производит вывод процедура `write`. Текущую позицию курсора можно получить, вызвав стандартные функции без параметров `WhereX` и `WhereY`. Для установки курсора в нужную позицию экрана используется стандартная процедура `GotoXY(x,y)`. Например, вызов процедуры `GotoXY(10,5)` поставит курсор в 10 колонку 5 строки экрана. Вот, теперь у нас есть минимальные средства для организации диалогового интерфейса с пользователем.

Хорошо организованный интерфейс должен обладать, по крайней мере, тремя свойствами. В каждый момент диалога

1. на экране должно быть всё необходимое, чтобы пользователь знал, что от него требуется со стороны программы;
2. на экране не должно быть ничего лишнего, не относящегося к этому этапу решения задачи;
3. когда возникла ошибка и о ней выдано сообщение, лучше, чтобы у пользователя был выбор, по крайней мере, из трёх возможностей:
   1). исправить ошибку (если это не фатальная ошибка);
   2). начать выполнения программы с начала;
   3). завершить выполнения программы.

Для организации диалогового интерфейса надо тщательно продумать сценарий (алгоритм) диалога с пользователем. Сначала надо очистить экран от ненужной информации и выдать пользователю приглашение к вводу данных. Человек лучше воспринимает информацию, выведенную ближе к центру экрана, чем в углах или на краях, поэтому выведем приглашение к вводу целого числа, например, начиная с 8 колонки 5 строки:

```
GotoXY(8,5); write('Введите X= ');
```

---

[1] За помощью по языку Free Pascal также можно обратиться на сайт http://freepascal.ru/download/book/doc_ref/index.html.

После этого программа переходит в *состояние ожидания* конца ввода, т.е. нажатия клавиши ENTER. Обратите внимание, что курсор сейчас находится после символа равно, а не в начале следующей строки, поэтому неверно использовать, как это часто делают учащиеся, процедуру `writeln` вместо процедуры `write`. Далее необходимо решить, что делать при вводе неправильных данных. Отведём, например, 25-ую строку экрана для выдачи аварийных сообщений (рис. 6.2).

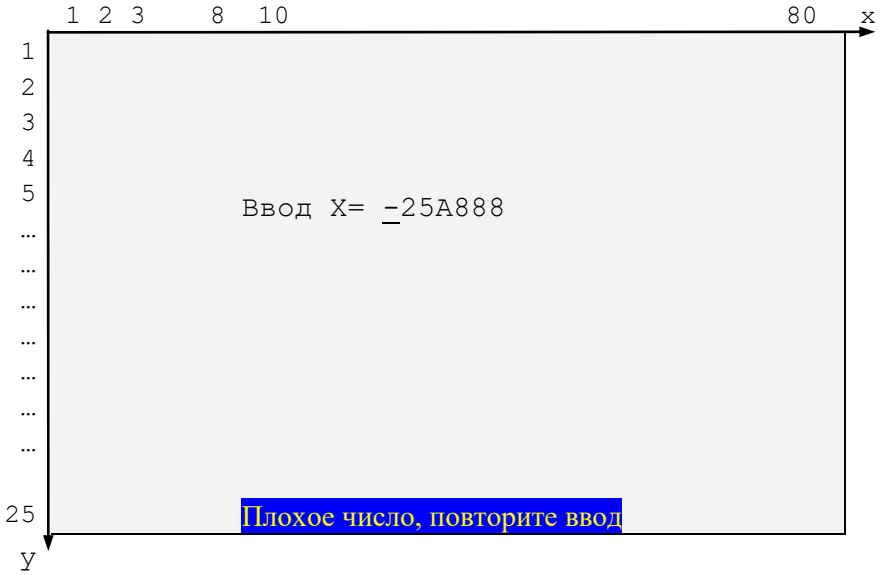

Рис. 6.2. Выдача аварийного сообщения.

Для привлечения внимания пользователя аварийная диагностика выведена бросающимися в глаза цветами (в данном случае жёлтыми буквами на синем фоне). После выдачи диагностики курсор снова находится в позиции ввода, причём неверно введённые данные *сохранены* на экране, чтобы пользователь видел допущенную при вводе ошибку. Далее следует решить вопрос, что делать, если пользователь ошибается при вводе снова и снова. Например, как и при вводе пин-кода кредитной карты, будем после третьей неудачной попытки, заканчивать программу с выдачей соответствующей аварийной диагностики.

Исходя из рассмотренных соображений, напишем первую версию программы, удовлетворяющую предусловию `Pred=true`.

```
  uses Crt,Windows; {$I-} { Блокируем ошибку ввода }
  var X,Y,OldColors,RetCode,InpNum: integer;
      Temp: int64; Sym: char;
begin
{ Заголовок окна программы и задачи }
SetConsoleTitle ('Ввод целого X, вывод Y=X+1');
repeat { Главный цикл }
  ClrScr; { Очистка экрана } InpNum:=0; { Число попыток ввода }
  repeat { Цикл ввода числа }
    GotoXY(8,5); Write('Введите X= '); readln(Temp);
    RetCode:=IOResult; { Получения кода возврата }
    InpNum:=InpNum+1;
    if RetCode<>0 then begin { Temp ∉ Z }
      OldColors:=TextAttr; { Запоминаем текущие цвета }
      TextAttr:=16*Blue+Yellow; { Цвета диагностики }
      GotoXY(8,25); Write('Плохое число, повторите ввод');
      TextAttr:=OldColors; { Восстановим текущие цвета }
      { ❶ }
    end else
    if (Temp<-MaxInt-1) or (Temp>=MaxInt) then begin
      OldColors:=TextAttr; TextAttr:=16*Black+LightRed;
      GotoXY(8,25); Write('Большое число, повторите ввод');
```

```
      TextAttr:=OldColors; RetCode:=1; {Был плохой ввод}
      {❶}
    end
  until (RetCode=0) or (InpNum=3);
  if (RetCode<>0) then begin
    OldColors:=TextAttr; TextAttr:=16*Blue+White;
    GotoXY(8,25); Write('Превышено число попыток ввода');
    TextAttr:=OldColors
  end else begin { Всё хорошо! }
    X:=Temp; Y:=X+1;
    GotoXY(8,10); Write('Ответ Y= ',Y);
    {❷}
    GotoXY(8,25); Write('Повторить Y/N ?');
    {❶}
    read(Sym)
  end
until (RetCode<>0) or (UpCase(Sym)<>'Y')
end.
```

В этой первой версии программы специально оставлено несколько ошибок в организации диалога с пользователем, предполагается, что они выявятся в процессе отладки. Во-первых, диагностики об ошибках и запрос повторного выполнении программы выводятся в одно место экрана, они имеют разную длину и разные цвета. В результате более длинный текст может «торчать» из-под более короткого текста. Во-вторых, после вывода правильного результата не удаляется выведенная ранее диагностика об ошибке, что, конечно, неправильно. И, наконец, не фиксируется исправление пользователем ошибки при вводе числа. Например, если сначала пользователь ошибся, ввёл

```
Введите X = 1234*
```

и получил диагностику

```
Плохое число, повторите ввод
```

затем исправил ошибку, введя правильное число 56

```
Введите X = 5634*
```

В этом случае программа выдаст

```
Ответ Y = 57
```

что выглядит весьма странно.

Для исправления этих ошибок в организации диалога можно использовать стандартную процедуру без параметров `ClrEol` из модуля `Crt`. Эта процедура, как можно догадаться из её имени, стирает все символы от позиции курсора до конца строки, при этом сам курсор не двигается. Места вставки этой процедуры помечены в предыдущей программе комментариями {❶}, их надо заменить на вызов процедуры `ClrEol`.

Для исправления введённого числа вместо комментария {❷} надо вставить строку

```
GotoXY(8,5); write('Введите X= ',X); ClrEol;
```

Не надо думать, что теперь в нашей программе не осталось ошибок в организации диалога с пользователем, но эти ошибки более тонкие. Например, что будет, если перед вводом числа несколько раз нажать клавишу ENTER? Кроме того, ошибка может возникнуть и при введении двух неправильных чисел подряд. Попробуйте исправить эти ошибки, предварительно хорошо изучив язык Free Pascal.

При реализации этой программы использовалась естественная концепция ввода данных пользователем. Эта концепция предполагает, что, в ответ на приглашение к вводу данных, пользователь может нажимать на любые символы, а также многократно стирать введённое и набирать заново. Другими словами, пользователь является полным хозяином положения до тех пор, пока не нажмёт клавишу ENTER, и только после этого введённые данные поступают на обработку в программу. Как было показано, при таком способе ввода возможны многочисленные ошибки.

Существует, однако, и альтернативная концепция ввода данных, иногда её называют методом «навязчивого сервиса». Суть этого метода ввода в том, чтобы не давать пользователю возможности

вводить неверные данные. Для реализации такого ввода используется метод ввода символов без буферизации, без эха и без контроля. При таком вводе код нажатого на клавиатуре символа немедленно (не ожидая нажатия клавиши ENTER) передаётся в программу, при этом он не анализируется на принадлежность к служебным (управляющим) символам и не выводится на экран автоматически.

В языке Free Pascal для такого ввода предназначена функция без параметров с именем `ReadKey`, которая возвращает символ, как только он будет нажат на клавиатуре. Здесь, однако, нужно сказать, что кроме основного набора символов, есть и вспомогательный набор (так как все символы в один основной 256-символьный алфавит не поместились). Например, во вспомогательный набор входят символы стрелочек, функциональных клавиш `F1-F12`, символы с комбинациями служебных клавиш `ALT` и `CTRL` и другие. При вводе символов из дополнительного алфавита они поступают во входной поток `input` в виде сразу двух символов (в двухбайтной кодировке), первым из которых является нулевой символ алфавита, а вторым – собственно символ из дополнительного алфавита. Например, при нажатии клавиши F1 в программу сначала поступают символ `chr(0)`, который удобно записывать как `#0`, а затем и символ, соответствующий клавише `F1` в дополнительном алфавите.

Итак, будем вводить из потока `input` символы, непрерывно проверяя введённую лексему целого числа на синтаксическую правильность и сами преобразовывать число во внутреннее машинное представление. По-прежнему заблокируем выдачу диагностики о плохом вводе директивой `{$I-}`, однако теперь, так как вводятся только символы, могут возникнуть лишь такие ошибки, как закрытый для ввода поток `input` и физическая неисправность устройства ввода. Кроме того, если поток `input` подключён к текстовому файлу,[1] то возможны также такие ошибки, как отсутствие соответствующего файла или невозможность читать из него. При возникновении всех таких ошибок будем выдавать универсальную диагностику «Ошибка ввода» и завершать выполнение программы.

Ниже показана возможная программа, реализующая новую стратегию ввода данных. Заметим, что в языке Free Pascal в правильном целом числе типа `integer` не более 5 десятичных цифр (в режиме `{$mode OBJFPC}` цифр там в два раза больше).

```
  uses Crt,Windows; {$I-} { Блокируем ошибку ввода }
  const Enter=#13; BS=#8; Beep=#7; Esc=#27;
  var X,Y,OldColors,RetCode,InpNum,NumSym,PosX: integer;
      Temp: Longint; Sym,Sign: char;
      BigNumber: boolean;
begin
{ Заголовок окна программы }
SetConsoleTitle ('Ввод целого X, вывод Y=X+1');
repeat { Главный цикл программы }
  ClrScr; { Очистка экрана } InpNum:=0; { Число попыток ввода }
  Sign:='+'; Temp:=0; BigNumber:=false;
  GotoXY(8,3); Write('Esc - выход из программы');
  repeat { Цикл попыток ввода числа }
    InpNum:=InpNum+1; GotoXY(8,5); Write('Введите X = ');
    Temp:=0; NumSym:=0; { Число введённых символов }
    repeat { Цикл ввода символов числа }
      Sym:=ReadKey; RetCode:=IOResult; { Код возврата }
      if RetCode<>0 then begin { Ошибка ввода }
        TextAttr:=16*Blue+Yellow; { Цвета диагностики }
        GotoXY(8,25);
        Write('Ошибка ввода, конец программы');
        ClrEol
      end else
      if (Sym in ['-','+']) and (NumSym=0) then
      begin { Знак числа }
```

[1] Для подключения потока `input` к текстовому файлу необходимо запускать программу на исполнение из командной строки в виде `C:>My_program.exe <My_File.txt`

```
        NumSym:=NumSym+1; Sign:=Sym; Write(Sym); { Эхо знака }
      end else
      if (Sym in ['0'..'9']) and (NumSym<5) then begin { Цифра }
        NumSym:=NumSym+1; Write(Sym); { Эхо цифры }
        Temp:=10*Temp+ord(Sym)-ord('0')
      end else
      if (Sym=BS) and (NumSym>0) then begin { Удалить символ }
        if NumSym=1 then Sign:='+'; { Возможно, знак числа }
        Temp:=Temp div 10; { Удаление цифры или знака числа }
        Write(BS,' ',BS); { Эхо удаления символа }
        NumSym:=NumSym-1
      end else
      if Sym=Esc then begin { Выход из программы }
        GotoXY(8,25); Write('Выход из программы');
        ReturnCode:=1; Sym:=ReadKey; { Задержка выхода }
      end else
      if (Sym<>Enter) or (NumSym=0) then begin { Плохой символ }
        if Sym=#0 then Sym:=ReadKey; { Дополнительный алфавит }
        PosX:=WhereX; { Позиция курсора }
        OldColors:=TextAttr; TextAttr:=16*Black+LightRed;
        GotoXY(8,25); Write('Плохой символ');
        Write(Beep); { Звуковой сигнал }
        TextAttr:=OldColors; ClrEol; GotoXY(PosX,5)
      end
    until (Sym=Enter) and (NumSym>0) or (RetCode<>0);
    if (RetCode=0) then begin { Проверка числа Temp>=0! }
      if Temp>=MaxInt then begin
        OldColors:=TextAttr; TextAttr:=16*Black+LightRed;
        GotoXY(8,25); Write('Большое число, повторите ввод');
        TextAttr:=OldColors; ClrEol; BigNumber:=True
      end
    end
  until (RetCode<>0) or (InpNum=3) or not BigNumber;
  if (RetCode=0) and (InpNum=3) then begin
    OldColors:=TextAttr; TextAttr:=16*Blue+White;
    GotoXY(8,25); Write('Превышено число попыток ввода');
    TextAttr:=OldColors
  end else
  if (RetCode=0) then begin { Всё хорошо! }
    X:=Temp; if Sign='-' then X:=-X;
    Y:=X+1; GotoXY(8,10); Write('Ответ Y= ',Y);
    GotoXY(8,5); Write('Введите X = ',X); ClrEol
    GotoXY(8,25); Write('Повторить Y/N ?'); ClrEol;
    Sym:=ReadKey
  end
until (RetCode<>0) or (Sym<>'Y') and (Sym<>'y')
end.
```

В приведённой выше программе специально допущена ошибка в алгоритме ввода числа, попытайтесь найти её и исправить.

Из приведённых примеров качественной реализации даже простейшей программы видно, насколько трудоёмок этот процесс. Принято считать, что, если для реализации только алгоритма решения задачи потрачена одна условная единица времени работы программиста, то для реализации полной программ (с контролем и интерфейсом) потребуется, по крайней мере, в четыре раза больше времени. Это показывает, насколько трудно написать *качественное* программное обеспечение.

Создание хорошего диалогового интерфейса достаточно сложная задача. Существуют многочисленные *инструментальные системы*, позволяющие облегчить и автоматизировать эту работу. В качестве примеров можно указать так называемые системы WYSIWYG (What You See Is What You Get), т.е. «что видишь на экране, то и получишь в работающей программе». Это, например, системы разработки диалоговых интерфейсов сайтов в Интернете. Эти системы позволяют конструировать интерфейс, располагая на экране все нужные элементы и задавая их свойства (например, тип значения для поля ввода).

Слово «интерфейс» означает отношение главный-подчинённый, имея в виду, что главным (ведущим) в диалоге является именно программа, в подчинённым (ведомым) – пользователь. В более сложном случае партнёры по диалогу являются равноправными, такие взаимоотношения регулируются правилами, называемыми протоколами. В качестве примеров можно указать дипломатические, а также сетевые протоколы.

Равноправным диалогом, как правило, не является и «беседа» с голосовым помощником, например, какими-нибудь Siri, Алисой или Марусей.

В языке Free Pascal в окне консоли можно открывать дополнительные текстовые окна. Для этого предназначена процедура

```
Window(x1,y1,x2,y2)
```

Здесь `(x1,y1)` – координаты левого верхнего, а `(x2,y2)` – правого нижнего углов нового окна. Координаты всегда отсчитываются от основного окна, которое обычно располагается с координатах `(1,1,80,25)`. Каждое новое текстовое окно располагается поверх существующего, безвозвратно затирая ту часть основного окна, на которой оно располагается. Никакого закрытия текущего окна и «возврата» к прежнему (родительскому) окну не предусмотрено. Так что это жалкое подобие «настоящих» графических окон Windows.

Вопросы и упражнения

Путь в тысячу ли начинается с первого шага.

Лао-Цзы, VI-V век до н.э.

1. Что такое правильная программа?
2. Что такое предусловие программы?
3. Как должна себя вести программа, если её предусловие не выполнено?
4. Почему следует требовать, чтобы программа имела тождественно истинное предусловие?
5. Что делается на всех трёх этапах ввода числа в программу?
6. Какая программа называется робастной?
7. Всегда ли у целого числа существует абсолютная величина?
8. Как можно определить, что введённый символ расположен в начале строки?
9. Что такое контроль входных данных?
10. Когда в программе нужен диалоговый интерфейс, а когда нет?
11. Что такое «навязчивый сервис» при вводе данных?
12. Что такое диалоговый интерфейс?
13. Когда для ввода чисел надо использовать процедуру `read`, а когда `readln` ?

---

[i] Для продвинутых читателей. Рассмотрим, например, такую часто используемую задачу, как поиск среднего арифметического двух (вещественных) чисел `y:=(x+y)/2`. Часто для решения этой задачи пишут такую программу.

```
  var x,y: real;
begin
  read(x,y); y:=(x+y)/2; write(y)
end.
```

Как люди уже опытные, мы сразу напишем предусловие этой программы.

```
Pred={x, y∈R ∧ (x+y)∈R}
```

Как видно, наша программа даёт правильный ответ только *для половины* возможных пар значений x и y 🙁. В то же время мы понимаем, что среднее арифметическое существует всегда! Так что нам нужно предусловие `Pred={x, y∈R}`. Немного подумав, можно предложить такую программу:

```
  var x,y: real;
begin
  read(x,y) ; y:=x/2+y/2; write(y)
end.
```

Новый алгоритм для нахождения среднего арифметического более трудный для ЭВМ, так как содержит две операции деления, вместо одной. Для современных компьютеров (как, впрочем, и для людей) умножение вещественных чисел занимает примерно в 3-5 раз больше времени, чем сложение, в деление – уже в 15-20 раз больше. Поэтому все компиляторы выражение `(x+y)/2` преобразуют в `(x+y)*0.5`, которое затем можно привести к «безопасному» виду `x*0.5+y*0.5`. Далее, большинство систем программирования позволяют решить эту задачу более эффективно, перейдя к старшему типу. Например, в языке Free Pascal можно написать оператор присваивание вида

```
y:=(extended(x)+y)*0.5
```

Здесь указано явное преобразование типа `real` в старший тип `extended`, что заставляет выполнять этот оператор по правилу

```
y:=real((extended(x)+extended(y))*0.5)
```

Преобразование типов поддерживается машинными командами и не увеличивает сложность алгоритма.

Проблема выхода промежуточного результата вычислений за допустимый диапазон настолько серьёзна, что большинство ЭВМ решают её на аппаратном уровне. Для этого, например, вещественные числа могут складываться на специальных регистрах длиной 80 бит (тип `extended`), так что оператор

```
y:=(x+y)*0.5
```

иожет быть откомпилирован в команды комьютера для выполнения по схеме

```
y:=real((extended(x)+extended(y))*0.5)
```

Сейчас, однако, на многих ЭВМ существуют так называемые векторные регистры, которые могут хранить не одно, а сразу несколько вещественных чисел, например, на регистры `YMM0-YMM15` длиной по 256 бит помещается сразу 4 числа типа `double` по 64 бита каждое:

YMM:	double	double	double	double

Тогда одна команда сложения будет складывать сразу по 4 пары чисел. Скорость обработки данных возрастает, но проблема выхода промежуточных результатов за допустимый диапазон снова встаёт в полный рост.

[ii] Для продвинутых читателей, знакомых с архитектурой ЭВМ. Можно считать, что для языков высокого уровня «физический мир» это уровень языка машины. Так вот, на уровне машинных команд, в операции обмена с помощью команд **xor** компилятором с языка высокого уровня неявно используется вспомогательная переменная на регистре, например на Ассемблере с помощью регистра RAX:

```
x dq   ?;
y dq   ?; var x,y: int64;
  mov  rax,x
  xor  y,rax; y:=y xor x;
  xor  rax,y
  mov  x,rax; x:=x xor y
  xor  y,rax; y:=y xor x
```

Впрочем, на языке Ассемблера операцию обмена значениями двух переменных с использованием регистра RAX можно записать и более компактно, например:

```
x dq   ?;
y dq   ?; var x,y: int64;
  mov  rax,x; rax:=x
  xchg y,rax; y:=rax; rax:=y
  mov  x,rax; x:=y
```

Может быть, однако, если переменные *уже* хранятся на регистрах, то можно обойтись без вспомогательной переменной? На первый взгляд, для 16-битных переменных это можно сделать с помощью команды **xchg**:

```
x equ  ax
y equ  bx
```

```
   ...
  xchg x,y
```

Однако, если рассмотреть команду обмена на уровне микроархитектуры ЭВМ, то оказывается, что она выполняется по правилу:

```
R1:=ax; ax:=bx; bx:=R1
```

где R1 – не адресуемый (служебный, невидимый программисту) регистр ЭВМ, т.е. по сути та же вспомогательная переменная.

Так что же, полная безнадёжность и без вспомогательной переменной не обойтись? Немного подумав, однако, можно предложить такой способ обмена содержимым, например, вагонов с рисом и пшеницей без использования промежуточного вагона. Можно брать по одному зерну риса и пшеницы и менять их местами, продолжая так до тех пор, пока все зерна не поменяются местами, здесь вспомогательная «переменная» совсем маленькая, равная одному зерну. Итак, задача решена! Правда, в физическом мире алгоритм получился очень затратный по проведённым операциям.

С точки зрения архитектуры ЭВМ таким «зерном» является один бит данных. Можно, например, предложить следующий алгоритм обмена значениями двух регистров по одному биту за раз:

```
   mov  cx,16; цикл на обмен 16 битами
   shl  ax,1; CF:=ax[15]
L: rcl  bx,1; bx[0]:=CF; CF:=bx[15]
   rcl  ax,1; ax[0]:=CF; CF:=ax[15]
   loop L
```

Впрочем, конвейеры современных ЭВМ решают задачу обмена значениями двух регистров предельно эффективно, просто путём переименования регистров, более подробно об этом говориться в курсе по архитектуре ЭВМ.