бых конечных множеств A и B семейство всех функций, отображающих A в B, является универсальным. Однако этот пример с практической точки зрения бесполезен: для запоминания случайной функции из этого семейства нужен массив, число элементов в котором равно числу элементов в множестве A. (А если мы можем себе позволить такой массив, то никакого хеширования не требуется!)

Более практичные примеры универсальных семейств могут быть построены с помощью несложных алгебраических конструкций. Через $\mathbb{Z}_p$ мы обозначаем множество вычетов по простому модулю p, т.е. $\{0, 1, \ldots, p-1\}$; арифметические операции в этом множестве выполняются по модулю p. Универсальное семейство образуют все линейные функционалы на $\mathbb{Z}_p^n$ со значениями в $\mathbb{Z}_p$. Более подробно, пусть $a_1, \ldots, a_n$ — произвольные элементы $\mathbb{Z}_p$; рассмотрим отображение

$$h : \langle x_1 \ldots x_n \rangle \mapsto a_1 x_1 + \ldots + a_n x_n$$

Мы получаем семейство из p^n отображений $\mathbb{Z}_p^n \to \mathbb{Z}_p$ параметризованное наборами $\langle a_1 \ldots a_n \rangle$.

11.2.5. Доказать, что это семейство является универсальным.

[Указание. Пусть x и y — различные точки пространства $\mathbb{Z}_p^n$. Какова вероятность того, что случайный функционал принимает на них одинаковые значения? Другими словами, какова вероятность того, что он равен нулю на их разности $x - y$? Ответ дается таким утверждением: пусть u — ненулевой вектор; тогда все значения случайного функционала на нём равновероятны.]

В следующей задаче множество $\mathbb{B} = \{0, 1\}$ рассматривается как множество вычетов по модулю 2.

11.2.6. Семейство всех линейных отображений из $\mathbb{B}^n$ в $\mathbb{B}^m$ является универсальным.

Родственные хешированию идеи неожиданно оказываются полезными в следующей ситуации (рассказал Д. Варсанофьев). Пусть мы хотим написать программу, которая обнаруживала (большинство) опечаток в тексте, но не хотим хранить список всех правильных словоформ. Предлагается поступить так: выбрать некоторое N и набор функций $f_1, \ldots, f_k$, отображающих русские слова в $1 \ldots N$. В массиве из N битов положим все биты равными нулю, кроме тех, которые являются значением какой-то функции набора на какой-то правильной словоформе. Теперь приближённый тест на правильность словоформы таков: проверить, что значения всех функций набора на этой словоформе попадают на места, занятые единицами. (Этот тест может не заметить некоторых ошибок, но все правильные словоформы будут одобрены.)

12. Представление множеств. Деревья. Сбалансированные деревья

12.1. Представление множеств с помощью деревьев

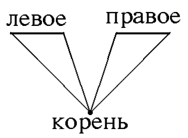
Полное двоичное дерево. Т-деревья

Нарисуем точку. Из нее проведём две стрелки (влево вверх и вправо вверх) в две другие точки. Из каждой из этих точек проведём по две стрелки и так далее. Полученную картинку (в n-ом слое будет 2^{n-1} точек) называют полным двоичным *деревом*. Нижнюю точку называют *корнем*. У каждой вершины есть два *сына* (две вершины, в которые идут стрелки) — *левый* и *правый*. У всякой вершины, кроме корня, есть единственный *отец*.

Пусть выбрано некоторое конечное множество вершин полного двоичного дерева, содержащее вместе с каждой вершиной и всех ее предков. Пусть на каждой вершине этого множества написано значение фиксированного типа Т (то есть задано отображение множества вершин в множество значений типа Т). То, что получится, будем называть Т-деревом. Множество всех Т-деревьев обозначим $\mathrm{Tree}(\mathsf{T})$.

Рекурсивное определение. Всякое непустое Т-дерево разбивается на три части: корень (несущий пометку из Т), левое и правое поддеревья (которые могут быть пустыми). Это разбиение устанавливает взаимно однозначное соответствие между множеством непустых Т-деревьев и произведением $\mathsf{T} \times \mathrm{Tree}(\mathsf{T}) \times \mathrm{Tree}(\mathsf{T})$. Обозначив через empty пустое дерево, можно написать

$$\mathrm{Tree}(\mathsf{T}) = \{\mathrm{empty}\} + \mathsf{T} \times \mathrm{Tree}(\mathsf{T}) \times \mathrm{Tree}(\mathsf{T}).$$

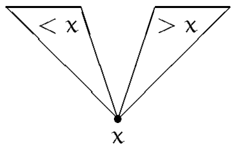


Поддеревья. Высота

Фиксируем некоторое Т-дерево. Для каждой его вершины x определено ее *левое поддерево* (левый сын вершины x и все его потомки), *правое* *поддерево* (правый сын вершины x и все его потомки) и *поддерево с корнем в* x (вершина x и все её потомки). Левое и правое поддеревья вершины x могут быть пустыми, а поддерево с корнем в x всегда непусто (содержит по крайней мере x). *Высотой* поддерева будем считать максимальную длину цепи $y_1 \ldots y_n$ его вершин, в которой y_{i+1} — сын y_i для всех i. (Высота пустого дерева равна нулю, высота дерева из одного корня — единице.)

Упорядоченные T-деревья

Пусть на множестве значений типа T фиксирован порядок. Назовем T-дерево *упорядоченным*, если выполнено такое свойство: для любой вершины x все пометки в ее левом поддереве меньше пометки в x, а все пометки в ее правом поддереве больше пометки в x.



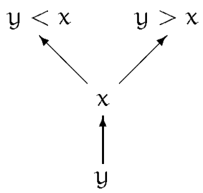
12.1.1. Доказать, что в упорядоченном дереве все пометки различны.

[Указание. Индукция по высоте дерева.]

Представление множеств с помощью деревьев

Каждое дерево будем считать представлением множества всех пометок на его вершинах. При этом одно и то же множество может иметь различные представления.

Благодаря упорядоченности каждый элемент может легко «найти своё место» в дереве: придя в какую-то вершину и сравнив себя с тем, кто там находится, элемент решает, идти ему налево или направо.



Начав с корня и двигаясь по этому правилу, он либо обнаружит, что такой элемент уже есть, либо найдет место, в котором он должен быть.

Всюду далее мы предполагаем, что на значениях типа T задан порядок, и рассматриваем только упорядоченные деревья.

Хранение деревьев в программе

Можно было бы сопоставить вершины полного двоичного дерева с числами $1, 2, 3, \ldots$ (считая, что левый сын n есть $2n$, правый сын n есть $2n+1$) и хранить пометки в массиве `val [1...]`. Однако этот способ неэкономен, поскольку тратится место на хранение пустых вакансий в полном двоичном дереве.

Более экономен такой способ. Введем три массива

```
val: array [1..n] of T;
left, right: array [1..n] of 0..n;
```

(`n` — максимальное возможное число вершин дерева) и переменную `root:0..n`. Каждая вершина хранимого T-дерева будет иметь номер — число от `1` до `n`. Разные вершины будут иметь разные номера. Пометка в вершине с номером `x` равна `val[x]`. Корень имеет номер `root`. Если вершина с номером `i` имеет сыновей, то их номера равны `left[i]` и `right[i]`. Отсутствующим сыновьям соответствует число `0`. Аналогичным образом значение `root=0` соответствует пустому дереву.

Для хранения дерева используется лишь часть массива; для тех `i`, которые свободны (не являются номерами вершин), значения `val[i]` безразличны. Нам будет удобно, чтобы все свободные числа были «связаны в список»: первое хранится в специальной переменной `free:0..n`, а следующее за `i` свободное число хранится в `left[i]`, так что свободны числа

```
free, left[free], left[left[free]],...
```

Для последнего свободного числа `i` значение `left[i]` равно `0`. Равенство `free=0` означает, что свободных чисел больше нет. (Замечание. Мы использовали для связывания свободных вершин массив `left`, но, конечно, с тем же успехом можно было использовать массив `right`.)

Вместо значения `0` (обозначающего отсутствие вершины) можно было бы воспользоваться любым другим числом вне `1..n`. Чтобы подчеркнуть это, будем вместо `0` использовать константу `null=0`.

12.1.2. Составить программу, определяющую, содержится ли элемент `t:T` в упорядоченном дереве (хранимом так, как только что описано).

Решение.

```
if root = null then begin
| ..не принадлежит
end else begin
| x := root;
| {инвариант: остается проверить наличие t в непустом
|   поддереве с корнем x}
| while ((t < val [x]) and (left [x] <> null)) or
| |     ((t > val [x]) and (right [x] <> null)) do begin
| | if t < val [x] then begin {left [x] <> null}
| | | x := left [x];
| | end else begin {t > val [x], right [x] <> null}
| | | x := right [x];
| | end;
| end;
| {либо t = val [x], либо t отсутствует в дереве}
| ..ответ = (t = val [x])
end;
```

12.1.3. Упростить решение, используя следующий трюк. Расширим область определения массива `val`, добавив ячейку с номером `null` и положим `val[null]=t`.

Решение.

```
val [null] := t;
x := root;
while t <> val [x] do begin
| if t < val [x] then begin
| | x := left [x];
| end else begin
| | x := right [x];
| end;
end;
..ответ: (x <> null).
```

12.1.4. Составить программу добавления элемента t в множество, представленное упорядоченным деревом (если элемент t уже есть, ничего делать не надо).

Решение. Определим процедуру `get_free (var i:integer)`, дающую свободное (не являющееся номером) число i и соответствующим образом корректирующую список свободных чисел.

```
procedure get_free (var i: integer);
begin
| {free <> null}
| i := free;
| free := left [free];
end;
```

С ее использованием программа приобретает вид:

```
if root = null then begin
| get_free (root);
| left [root] := null; right [root] := null;
| val [root] := t;
end else begin
| x := root;
| {инвариант: осталось добавить t к непустому поддереву с
|  корнем в x}
| while ((t < val [x]) and (left [x] <> null)) or
```

```
      ((t > val [x]) and (right [x] <> null)) do begin
    if t < val [x] then begin
      x := left [x];
    end else begin {t > val [x]}
      x := right [x];
    end;
  end;
  if t <> val [x] then begin {t нет в дереве}
    get_free (i);
    left [i] := null; right [i] := null;
    val [i] := t;
    if t < val [x] then begin
      left [x] := i;
    end else begin {t > val [x]}
      right [x] := i;
    end;
  end;
end;
```

12.1.5. Составить программу удаления элемента t из множества, представленного упорядоченным деревом (если его там нет, ничего делать не надо).

Решение.

```
if root = null then begin
  {дерево пусто, ничего делать не надо}
end else begin
  x := root;
  {осталось удалить t из поддерева с корнем в x; поскольку
   это может потребовать изменений в отце x, введем
   переменные  father: 1..n и direction: (l, r);
   поддерживаем такой инвариант: если x не корень, то father
   - его отец, а direction равно l или r в зависимости от
   того, левым или правым сыном является x}
  while ((t < val [x]) and (left [x] <> null)) or
        ((t > val [x]) and (right [x] <> null)) do begin
```

```
    if t < val [x] then begin
      father := x; direction := l;
      x := left [x];
    end else begin {t > val [x]}
      father := x; direction := r;
      x := right [x];
    end;
  end;
  {t = val [x] или t нет в дереве}
  if t = val [x] then begin
    ..удаление вершины x  с отцом father и
      направлением direction
  end;
end;
```

Удаление вершины использует процедуру

```
  procedure make_free (i: integer);
  begin
    left [i] := free;
    free := i;
  end;
```

Она включает число i в список свободных. При удалении различаются 4 случая в зависимости от наличия или отсутствия сыновей у удаляемой вершины.

```
if (left [x] = null) and (right [x] = null) then begin
  {x - лист, т.е. не имеет сыновей}
  make_free (x);
  if x = root then begin
    root := null;
  end else if direction = l then begin
    left [father] := null;
  end else begin {direction = r}
    right [father] := null;
  end;
```

```
end else if (left[x]=null) and (right[x] <> null) then begin
| {x удаляется, а right [x] занимает место x}
| make_free (x);
| if x = root then begin
| | root := right [x];
| end else if direction = l then begin
| | left [father] := right [x];
| end else begin {direction = r}
| | right [father] := right [x];
| end;
end else if (left[x] <> null) and (right[x]=null) then begin
| ..симметрично
end else begin {left [x] <> null, right [x] <> null}
| ..удалить вершину с двумя сыновьями
end;
```

Удаление вершины с двумя сыновьями нельзя сделать просто так, но ее можно предварительно поменять с вершиной, пометка на которой является непосредственно следующим (в порядке возрастания) элементом за пометкой на `x`.

```
  y := right [x];
  father := x; direction := r;
  {теперь father и direction относятся к вершине y}
  while left [y] <> null do begin
  | father := y; direction := l;
  | y := left [y];
  end;
  {val [y] - минимальная из пометок, больших val [x],
   y не имеет левого сына}
  val [x] := val [y];
  ..удалить вершину y (как удалять вершину, у которой нет
    левого сына, мы уже знаем)
```

12.1.6. Упростить программу удаления, заметив, что некоторые случаи (например, первые два из четырёх) можно объединить.

12.1.7. Использовать упорядоченные деревья для представления функций, область определения которых — конечные множества значений типа `T`, а значения имеют некоторый тип `U`. Операции: вычисление значения на данном аргументе, изменение значения на данном аргументе, доопределение функции на данном аргументе, исключение элемента из области определения функции.

Решение. Делаем как раньше, добавив ещё один массив

```
func_val: array [1..n] of U;
```

если `val[x] = t`, `func_val[x] = u`, то значение хранимой функции на `t` равно `u`.

12.1.8. Предположим, что необходимо уметь также отыскивать `k`-ый элемент множества (в порядке возрастания), причем количество действий должно быть не более $C\cdot$(высота дерева). Какую дополнительную информацию надо хранить в вершинах дерева?

Решение. В каждой вершине будем хранить число всех её потомков. Добавление и исключение вершины требует коррекции лишь на пути от корня к этой вершине. В процессе поиска `k`-ой вершины поддерживается такой инвариант: искомая вершина является `s`-ой вершиной поддерева с корнем в `x` (здесь `s` и `x` — переменные).

Оценка количества действий

Для каждой из операций (проверки, добавления и исключения) количество действий не превосходит $C\cdot$(высота дерева). Для «ровно подстриженного» дерева (когда все листья на одной высоте) высота по порядку величины равна логарифму числа вершин. Однако для кривобокого дерева всё может быть гораздо хуже: в наихудшем случае все вершины образуют цепь и высота равна числу вершин. Так случится, если элементы множества добавляются в возрастающем или убывающем порядке. Можно доказать, однако, что при добавлении элементов «в случайном порядке» средняя высота дерева будет не больше $C\log$(число вершин). Если этой оценки «в среднем» мало, необходимы дополнительные действия по поддержанию «сбалансированности» дерева. Об этом смотри в следующем пункте.

12.2. Сбалансированные деревья

Дерево называется *сбалансированным* (или АВЛ-деревом в честь изобретателей этого метода Г.М. Адельсона-Вельского и Е.М. Ландиса), если для любой его вершины высоты левого и правого поддеревьев этой вершины отличаются не более чем на 1. (В частности, когда одного из сыновей нет, другой — если он есть — обязан быть листом.)

12.2.1. Найти минимальное и максимальное возможное количество вершин в сбалансированном дереве высоты n.

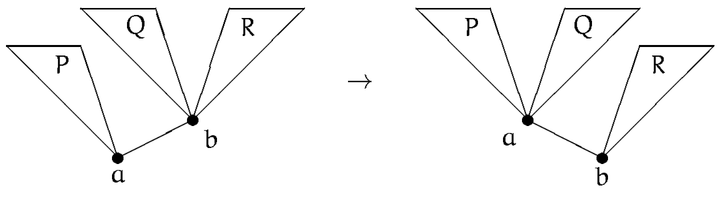
Решение. Максимальное число вершин равно $2^n - 1$. Если m_n — минимальное число вершин, то, как легко видеть, $m_{n+2} = 1 + m_n + m_{n+1}$, откуда $m_n = \Phi_{n+2} - 1$ (Φ_n — n-ое число Фибоначчи, $\Phi_1 = 1$, $\Phi_2 = 1$, $\Phi_{n+2} = \Phi_n + \Phi_{n+1}$).

12.2.2. Доказать, что сбалансированное дерево с n вершинами имеет высоту не больше $C \log n$ для некоторой константы C, не зависящей от n.

Решение. Индукцией по n легко доказать, что $\Phi_{n+2} \geqslant a^n$, где a — больший корень квадратного уравнения $a^2 = 1 + a$, то есть $a = (\sqrt{5}+1)/2$. Остается воспользоваться предыдущей задачей.

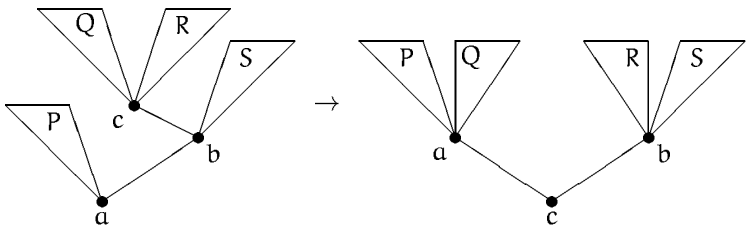
Вращения

Мы хотим восстанавливать сбалансированность дерева после включения и удаления элементов. Для этого необходимы какие-то преобразования дерева, не меняющие множества пометок на его вершинах и не нарушающие упорядоченности, но способствующие лучшей сбалансированности. Опишем несколько таких преобразований.



Пусть вершина a имеет правого сына b. Обозначим через P левое поддерево вершины a, через Q и R — левое и правое поддеревья вершины b. Упорядоченность дерева требует, чтобы $P < a < Q < b < R$ (точнее следовало бы сказать «любая пометка на P меньше пометки на a», «пометка на a меньше любой пометки на Q» и т.д., но мы позволим себе этого не делать). Точно того же требует упорядоченность дерева с корнем b, его левым сыном a, в котором P и Q — левое и правое поддеревья a, R — правое поддерево b. Поэтому первое дерево можно преобразовать во второе, не нарушая упорядоченности. Такое преобразование назовем *малым правым вращением* (правым — поскольку существует симметричное, левое, малым — поскольку есть и большое, которое мы сейчас опишем).

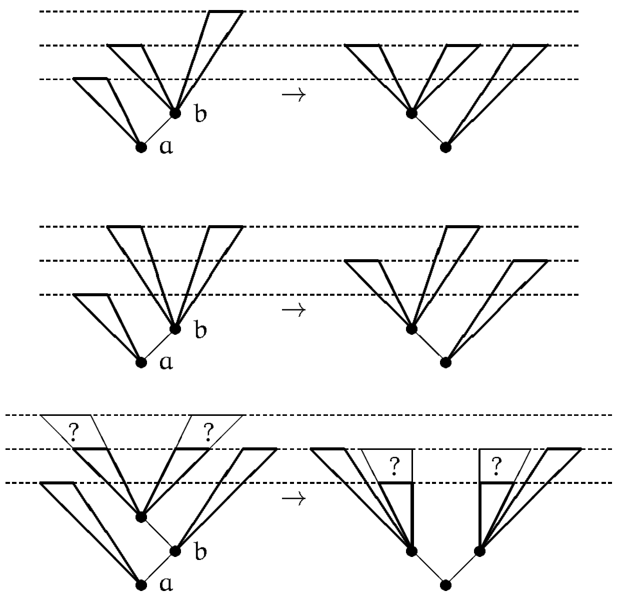
Пусть b — правый сын a, c — левый сын b, P — левое поддерево a, Q и R — левое и правое поддеревья c, S — правое поддерево b. Тогда $P < a < Q < c < R < b < S$.



Такой же порядок соответствует дереву с корнем c, имеющим левого сына a и правого сына b, для которого P и Q — поддеревья вершины a, а R и S — поддеревья вершины b. Соответствующее преобразование будем называть *большим правым вращением*. (Аналогично определяется симметричное ему *большое левое вращение*.)

12.2.3. Дано дерево, сбалансированное всюду, кроме корня, в котором разница высот равна 2 (т.е. левое и правое поддеревья корня сбалансированы и их высоты отличаются на 2). Доказать, что оно может быть превращено в сбалансированное одним из четырёх описанных преобразований, причем высота его останется прежней или уменьшится на 1.

Решение. Пусть более низким является, например, левое поддерево, и его высота равна k. Тогда высота правого поддерева равна $k+2$. Обозначим корень через a, а его правого сына (он обязательно есть) через b. Рассмотрим левое и правое поддеревья вершины b. Одно из них обязательно имеет высоту $k+1$, а другое может иметь высоту k или $k+1$ (меньше k быть не может, так как поддеревья сбалансированы). Если высота левого поддерева равна $k+1$, а правого — k, то потребуется большое правое вращение; в остальных случаях помогает малое. Вот как выглядят три случая балансировки дерева:



12.2.4. В сбалансированное дерево добавили или из него удалили лист. Доказать, что можно восстановить сбалансированность с помощью нескольких вращений, причем их число не больше высоты дерева.

Решение. Будем доказывать более общий факт:

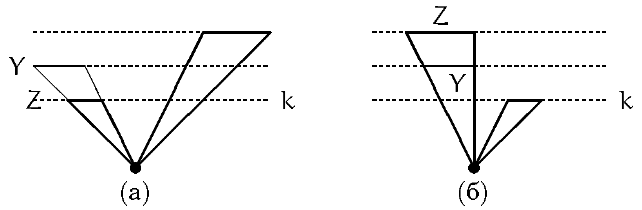
Лемма. Если в сбалансированном дереве X одно из его поддеревьев Y заменили на сбалансированное дерево Z, причем высота Z отличается от высоты Y не более чем на 1, то полученное такой «прививкой» дерево можно превратить в сбалансированное вращениями (причем количество вращений не превосходит высоты, на которой делается прививка).

Частным случаем прививки является замена пустого поддерева на лист или наоборот, так что достаточно доказать эту лемму.

Доказательство леммы. Индукция по высоте, на которой делается прививка. Если она происходит в корне (заменяется все дерево целиком), то все очевидно («привой» сбалансирован по условию). Пусть заменяется некоторое поддерево, например, левое поддерево некоторой вершины x. Возможны два случая.

1) После прививки сбалансированность в вершине x не нарушилась (хотя, возможно, нарушилась сбалансированность в предках x: высота поддерева с корнем в x могла измениться). Тогда можно сослаться на предположение индукции, считая, что мы прививали целиком поддерево с корнем в x.

2) Сбалансированность в x нарушилась. При этом разница высот равна 2 (больше она быть не может, так как высота Z отличается от высоты Y не более чем на 1). Разберем два варианта.



а) Выше правое (не заменявшееся) поддерево вершины x. Пусть высота левого (т.е. Z) равна k, правого — k+2. Высота старого левого поддерева вершины x (т.е. Y) была равна $k+1$. Поддерево с корнем x имело в исходном дереве высоту $k+3$, и эта высота не изменилась после прививки.

По предыдущей задаче вращение преобразует поддерево с корнем в x в сбалансированное поддерево высоты k+2 или k+3. То есть высота поддерева с корнем x — в сравнении с его прежней высотой — не изменилась или уменьшилась на 1, и мы можем воспользоваться предположением индукции.

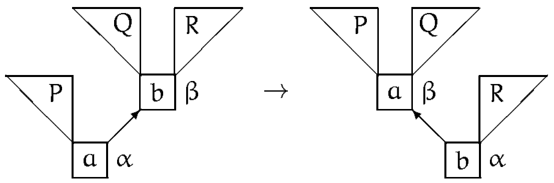
б) Выше левое поддерево вершины x. Пусть высота левого (т.е. Z) равна $k+2$, правого — k. Высота старого левого поддерева (т.е. Y) была равна $k+1$. Поддерево с корнем x в исходном дереве X имело высоту $k+2$, после прививки она стала равна $k+3$. После подходящего вращения (см. предыдущую задачу) поддерево с корнем в x станет сбалансированным, его высота будет равна $k+2$ или $k+3$, так что изменение высоты по сравнению с высотой поддерева с корнем x в дереве X не превосходит 1 и можно сослаться на предположение индукции.

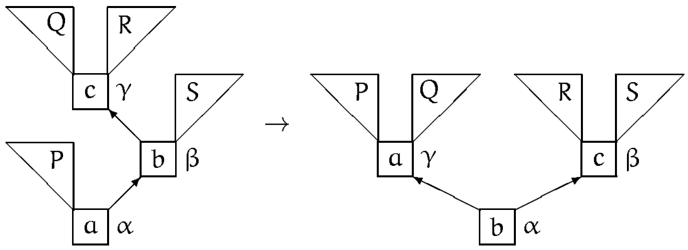
12.2.5. Составить программы добавления и удаления элементов, сохраняющие сбалансированность. Число действий не должно превосходить $C \cdot$ (высота дерева). Разрешается хранить в вершинах дерева дополнительную информацию, необходимую при балансировке.

Решение. Будем хранить для каждой вершины разницу между высотой ее правого и левого поддеревьев:

$$\texttt{diff [i]} = (\text{высота правого поддерева вершины } \texttt{i}) - (\text{высота левого поддерева вершины } \texttt{i}).$$

Нам потребуются четыре процедуры, соответствующие большим и малым правым и левым вращениям. Но вначале два замечания. (1) Нам нужно, чтобы при вращении поддерева номер его корня не менялся. (В противном случае потребовалось бы корректировать информацию в отце корня, что нежелательно.) Этого можно достичь, так как номера вершин дерева можно выбирать независимо от их значений. (На картинках номер указан сбоку от вершины, а значение — внутри.)





(2) После преобразований мы должны также изменить соответственно значения в массиве diff. Для этого достаточно знать высоты деревьев $P, Q, \ldots$ с точностью до константы, поэтому можно предполагать, что одна из высот равна нулю.

Вот процедуры вращений:

```
procedure SR (a:integer); {малое правое вращение с корнем a}
| var b: 1..n; val_a,val_b: T; h_P,h_Q,h_R: integer;
begin
| b := right [a]; {b <> null}
| val_a := val [a]; val_b := val [b];
| h_Q := 0; h_R := diff[b]; h_P := (max(h_Q,h_R)+1)-diff[a];
| val [a] := val_b; val [b] := val_a;
| right [a] := right [b] {поддерево R}
| right [b] := left [b] {поддерево Q}
| left [b] := left [a] {поддерево P}
```

```
  left [a] := b;
  diff [b] := h_Q - h_P;
  diff [a] := h_R - (max (h_P, h_Q) + 1);
end;

procedure BR(a:integer);{большое правое вращение с корнем a}
  var b,c: 1..n; val_a,val_b,val_c: T;
      h_P,h_Q,h_R,h_S: integer;
begin
  b := right [a]; c := left [b]; {,c <> null}
  val_a := val [a]; val_b := val [b]; val_c := val [c];
  h_Q := 0; h_R := diff[c]; h_S := (max(h_Q,h_R)+1)+diff[b];
  h_P := 1 + max (h_S, h_S-diff[b]) - diff [a];
  val [a] := val_c; val [c] := val_a;
  left [b] := right [c] {поддерево R}
  right [c] := left [c] {поддерево Q}
  left [c] := left [a] {поддерево P}
  left [a] := c;
  diff [b] := h_S - h_R;
  diff [c] := h_Q - h_P;
  diff [a] := max (h_S, h_R) - max (h_P, h_Q);
end;
```

Левые вращения (большое и малое) записываются симметрично.

Процедуры добавления и удаления элементов пишутся как раньше, но только добавление и удаление должно сопровождаться коррекцией массива `diff` и восстановлением сбалансированности.

При этом используется процедура с такими свойствами:

> **дано:** левое и правое поддеревья вершины с номером `a` сбалансированы, в самой вершине разница высот не больше 2, в поддереве с корнем `a` массив `diff` заполнен правильно;
>
> **надо:** поддерево с корнем `a` сбалансировано и массив `diff` соответственно изменен, `d` — изменение его высоты (равно `0` или `-1`); в остальной части всё осталось как было — в частности, значения `diff`

```
procedure balance (a: integer; var d: integer);
begin {-2 <= diff[a] <= 2}
| if diff [a] = 2 then begin
| | b := right [a];
| | if diff [b] = -1 then begin
| | | BR (a); d := -1;
| | end else if diff [b] = 0 then begin
| | | SR (a); d := 0;
| | end else begin {diff [b] = 1}
| | | SR (a); d := - 1;
| | end;
| end else if diff [a] = -2 then begin
| | b := left [a];
| | if diff [b] = 1 then begin
| | | BL (a); d := -1;
| | end else if diff [b] = 0 then begin
| | | SL (a); d := 0;
| | end else begin {diff [b] = -1}
| | | SL (a); d := - 1;
| | end;
| end else begin {-2 < diff [a] < 2, ничего делать не надо}
| | d := 0;
| end;
end;
```

Восстановление сбалансированности требует движения от листьев к корню, поэтому будем хранить в стеке путь от корня к рассматриваемой в данный момент вершине. Элементами стека будут пары ⟨вершина, направление движения из неё⟩, т.е. значения типа

```
record
| vert: 1..n; {вершина}
| direction : (l, r); {l - левое, r - правое}
end;
```

Программа добавления элемента t теперь выглядит так:

```
if root = null then begin
| get_free (root);
| left[root] := null; right[root] := null; diff[root] := 0;
| val[root] := t;
end else begin
| x := root; ..сделать стек пустым
| {инвариант: осталось добавить t к непустому поддереву с
|  корнем в x; стек содержит путь к x}
| while ((t < val [x]) and (left [x] <> null)) or
| |     ((t > val [x]) and (right [x] <> null)) do begin
| | if t < val [x] then begin
| | | ..добавить в стек пару <x, l>
| | | x := left [x];
| | end else begin {t > val [x]}
| | | ..добавить в стек пару <x, r>
| | | x := right [x];
| | end;
| end;
| if t <> val [x] then begin {t нет в дереве}
| | get_free (i); val [i] := t;
| | left [i] := null; right [i] := null; diff [i] := 0;
| | if t < val [x] then begin
| | | ..добавить в стек пару <x, l>
| | | left [x] := i;
| | end else begin {t > val [x]}
| | | ..добавить в стек пару <x, r>
| | | right [x] := i;
| | end;
| | d := 1;
| | {инвариант: стек содержит путь к изменившемуся
| |  поддереву, высота  которого увеличилась по
| |  сравнению с высотой в исходном дереве
| |  на d (=0 или 1); это поддерево  сбалансировано;
| |  значения diff для его вершин правильны; в
| |  остальном дереве  все  осталось  как  было -
```

```
      в частности,значения diff}
    while (d <> 0) and ..стек непуст do begin {d = 1}
      ..взять из стека пару в <v, direct>
      if direct = l then begin
        if diff [v] = 1 then begin
          c := 0;
        end else begin
          c := 1;
        end;
        diff [v] := diff [v] - 1;
      end else begin {direct = r}
        if diff [v] = -1 then begin
          c := 0;
        end else begin
          c := 1;
        end;
        diff [v] := diff [v] + 1;
      end;
      {c = изменение высоты поддерева с корнем в v по
       сравнению с исходным деревом; массив diff
       содержит правильные значения для этого поддерева;
       возможно нарушение сбалансированности в v}
      balance (v, d1); d := c + d1;
    end;
  end;
end;
```

Легко проверить, что значение d может быть равно только 0 или 1 (но не -1): если c=0, то diff[v]=0 и балансировка не производится.

Программа удаления строится аналогично. Ее основной фрагмент таков:

```
  {инвариант: стек содержит путь к изменившемуся поддереву,
   высота которого изменилась по сравнению с высотой в
   исходном дереве на d (=0 или -1); это поддерево
   сбалансировано; значения diff для его вершин правильны;
```

```
   в остальном дереве все осталось как было -
   в частности, значения diff}
  while (d <> 0) and ..стек непуст do begin
  | {d = -1}
  | ..взять из стека пару в <v, direct>
  | if direct = l then begin
  | | if diff [v] = -1 then begin
  | | | c := -1;
  | | end else begin
  | | | c := 0;
  | | end;
  | | diff [v] := diff [v] + 1;
  | end else begin {direct = r}
  | | if diff [v] = 1 then begin
  | | | c := -1;
  | | end else begin
  | | | c := 0;
  | | end;
  | | diff [v] := diff [v] - 1;
  | end;
  | {c = изменение высоты поддерева с корнем в v по срав-
  |  нению с исходным деревом; массив diff содержит
  |  правильные значения для этого поддерева;
  |  возможно нарушение сбалансированности в v}
  | balance (v, d1);
  | d := c + d1;
  end;
```

Легко проверить, что значение d может быть равно только 0 или -1 (но не -2): если c=-1, то diff[v]=0 и балансировка не производится.

Отметим также, что наличие стека делает излишними переменные father и direction (их роль теперь играет вершина стека).

12.2.6. Доказать, что при добавлении элемента

(а) второй из трех случаев балансировки (см. рисунок на с. 214) невозможен;

(б) полная балансировка требует не более одного вращения (после чего все дерево становится сбалансированным), в то время как при удалении элемента может понадобиться много вращений.

Замечание. Мы старались записать программы добавления и удаления так, чтобы они были как можно более похожими друг на друга. Используя специфику каждой из них, можно многое упростить.

Существуют и другие способы представления множеств, гарантирующие число действий порядка $\log n$ на каждую операцию. Опишем один из них (называемый *Б-деревьями*).

До сих пор каждая вершина содержала один элемент хранимого множества. Этот элемент служил границей между левым и правым поддеревом. Будем теперь хранить в вершине $k \geqslant 1$ элементов множества (число k может меняться от вершины к вершине, а также при добавлении и удалении новых элементов, см. далее). Эти k элементов служат разделителями для $k + 1$ поддерева. Пусть фиксировано некоторое число $t \geqslant 1$. Будем рассматривать деревья, обладающие такими свойствами:

1) Каждая вершина содержит от t до $2t$ элементов (за исключением корня, который может содержать любое число элементов от 0 до $2t$).

2) Вершина с k элементами либо имеет $k + 1$ сына, либо не имеет сыновей вообще (является *листом*).

3) Все листья находятся на одной и той же высоте.

Добавление элемента происходит так. Если лист, в который он попадает, неполон (т.е. содержит менее $2t$ элементов), то нет проблем. Если он полон, то $2t + 1$ элемент (все элементы листа и новый элемент) разбиваем на два листа по t элементов и разделяющий их серединный элемент. Этот серединный элемент надо добавить в вершину предыдущего уровня. Это возможно, если в ней менее $2t$ элементов. Если и она полна, то её разбивают на две, выделяют серединный элемент и т.д. Если в конце концов мы захотим добавить элемент в корень, а он окажется полным, то корень расщепляется на две вершины, а высота дерева увеличивается на 1.

Удаление элемента, находящегося не в листе, сводится к удалению непосредственно следующего за ним, который находится в листе. Поэтому достаточно научиться удалять элемент из листа. Если лист при этом становится слишком маленьким, то его можно пополнить за счет соседнего листа — если только и он не имеет минимально возможный размер t. Если же оба листа имеют размер t, то на них вместе $2t$ элементов, вместе с разделителем — $2t+1$. После удаления одного элемента остается $2t$ элементов — как раз на один лист. Если при этом вершина предыдущего уровня становится меньше нормы, процесс повторяется и т.д.

12.2.7. Реализовать описанную схему хранения множеств, убедившись, что она также позволяет обойтись $C\log n$ действий для операций включения, исключения и проверки принадлежности.

12.2.8. Можно определять сбалансированность дерева иначе: требовать, чтобы для каждой вершины ее левое и правое поддеревья имели не слишком сильно отличающиеся количества вершин. (Преимущество такого определения состоит в том, что при вращениях не нарушается сбалансированность в вершинах, находящихся ниже точки вращения.) Реализовать на основе этой идеи способ хранения множеств, гарантирующий оценку в $C\log n$ действий для включения, удаления и проверки принадлежности.

[Указание. Он также использует большие и малые вращения. Подробности см. в книге Рейнгольда, Нивергельта и Део «Комбинаторные алгоритмы».)]

13. Контекстно-свободные грамматики

13.1. Общий алгоритм разбора

Чтобы определить то, что называют *контекстно-свободной грамматикой* (КС-грамматикой), надо:

- указать конечное множество A, называемое *алфавитом*; его элементы называют *символами*; конечные последовательности символов называют *словами* (в данном алфавите);
- разделить все символы алфавита A на две группы: *терминальные* («окончательные») и *нетерминальные* («промежуточные»);
- выбрать среди нетерминальных символов один, называемый *начальным*;
- указать конечное число *правил* грамматики, каждое из которых должно иметь вид $K \to X$, где K — некоторый нетерминальный символ, а X — слово (в него могут входить и терминальные, и нетерминальные символы).

Пусть фиксирована КС-грамматика (мы часто будем опускать приставку «КС-», так как других грамматик у нас не будет). *Выводом* в этой грамматике называется последовательность слов $X_0, X_1, \ldots, X_n$, в которой X_0 состоит из одного символа, и этот символ — начальный, а X_{i+1} получается из X_i заменой некоторого нетерминального символа K на слово X по одному из правил грамматики. Слово, составленное из терминальных символов, называется *выводимым*, если существует вывод, который им кончается. Множество всех выводимых слов (из терминальных символов) называется *языком, порождаемым данной грамматикой*.

В этой и следующей главе нас будет интересовать такой вопрос: дана КС-грамматика; построить алгоритм, который по любому слову проверяет, выводимо ли оно в этой грамматике.

Пример 1. Алфавит:

```
( ) [ ] E
```

(четыре терминальных символа и один нетерминальный символ `E`). Начальный символ: `E`. Правила:

```
E → (E)
E → [E]
E → EE
E →
```

(в последнем правиле справа стоит пустое слово).

Примеры выводимых слов:

```
         (пустое слово)
()
([])
()[([])]
[()[]()[]]
```

Примеры невыводимых слов:

```
(
)(
(]
([)]
```

Эта грамматика встречалась в разделе 6 (где выводимость в ней проверялась с помощью стека).

Пример 2. Другая грамматика, порождающая тот же язык:

Алфавит: `( ) [ ] T E`

Правила:

```
E →
E → TE
T → (E)
T → [E]
```

Начальным символом во всех приводимых далее примерах будем считать символ, стоящий в левой части первого правила (в данном случае это символ `E`), не оговаривая этого особо.

Для каждого нетерминального символа можно рассмотреть множество всех слов из терминальных символов, которые из него выводятся (аналогично тому, как это сделано для начального символа в определении выводимости в грамматике). Каждое правило грамматики можно рассматривать как свойство этих множеств. Покажем это на примере только что приведенной грамматики. Пусть T и E — множества слов (из скобок), выводимых из нетерминалов `T` и `E` соответственно. Тогда правилам грамматики соответствуют такие свойства:

`E` $\to$	E содержит пустое слово
`E` $\to$ `TE`	если слово A принадлежит T, а слово B принадлежит E, то слово AB принадлежит E
`T` $\to$ `[E]`	если A принадлежит E, то слово $[\mathsf{A}]$ принадлежит T
`T` $\to$ `(E)`	если A принадлежит E, то слово (A) принадлежит T

Сформулированные свойства множеств E, T не определяют эти множества однозначно (например, они остаются верными, если в качестве E и T взять множество всех слов). Однако можно доказать, что множества, задаваемые грамматикой, являются минимальными среди удовлетворяющих этим условиям.

13.1.1. Сформулируйте точно и докажите это утверждение для произвольной контекстно-свободной грамматики.

13.1.2. Постройте грамматику, в которой выводимы слова

(а) `00..0011..11` (число нулей равно числу единиц);

(б) `00..0011..11` (число нулей вдвое больше числа единиц);

(в) `00..0011..11` (число нулей больше числа единиц);

(и только они).

13.1.3. Доказать, что не существует КС-грамматики, в которой были бы выводимы слова вида `00..0011..1122..22`, в которых числа нулей, единиц и двоек равны, и только они.

[Указание. Докажите следующую лемму о произвольной КС-грамматике: для любого достаточно длинного слова F, выводимого в этой грамматике, существует такое его представление в виде $ABCDE$, что любое слово вида $AB\ldots BCD\ldots DE$, где B и D повторены одинаковое число раз, также выводимо в этой грамматике. (Это можно установить, найдя нетерминальный символ, оказывающийся своим собственным «наследником» в процессе вывода.)]

Нетерминальный символ можно рассматривать как «родовое имя» для выводимых из него слов. В следующем примере для наглядности в качестве нетерминальных символов использованы фрагменты русских слов, заключенные в угловые скобки. (С точки зрения грамматики каждый такой фрагмент — один символ!)

Пример 3. Алфавит:

терминалы: `+ * ( ) x`
нетерминалы: ⟨выр⟩ ⟨оствыр⟩ ⟨слаг⟩ ⟨остслаг⟩ ⟨множ⟩
правила:

$$
\begin{array}{rcl}
\langle\text{выр}\rangle & \to & \langle\text{слаг}\rangle\ \langle\text{оствыр}\rangle \\
\langle\text{оствыр}\rangle & \to & \texttt{+}\ \langle\text{выр}\rangle \\
\langle\text{оствыр}\rangle & \to & \\
\langle\text{слаг}\rangle & \to & \langle\text{множ}\rangle\ \langle\text{остслаг}\rangle \\
\langle\text{остслаг}\rangle & \to & \texttt{*}\ \langle\text{слаг}\rangle \\
\langle\text{остслаг}\rangle & \to & \\
\langle\text{множ}\rangle & \to & \texttt{x} \\
\langle\text{множ}\rangle & \to & \texttt{(}\ \langle\text{выр}\rangle\ \texttt{)}
\end{array}
$$

Согласно этой грамматике, выражение ⟨выр⟩ — это последовательность слагаемых ⟨слаг⟩, разделенных плюсами, слагаемое — это последовательность множителей ⟨множ⟩, разделенных звёздочками (знаками умножения), а множитель — это либо буква `x`, либо выражение в скобках.

13.1.4. Приведите пример другой грамматики, задающей тот же язык.

Ответ. Вот один из вариантов:

$$\begin{array}{lcl} \langle\text{выр}\rangle & \to & \langle\text{выр}\rangle + \langle\text{выр}\rangle \\ \langle\text{выр}\rangle & \to & \langle\text{выр}\rangle * \langle\text{выр}\rangle \\ \langle\text{выр}\rangle & \to & \mathtt{x} \\ \langle\text{выр}\rangle & \to & (\ \langle\text{выр}\rangle\ ) \end{array}$$

Эта грамматика хоть и проще, но в некоторых отношениях хуже, о чём мы ещё будем говорить.

13.1.5. Дана произвольная КС-грамматика. Построить алгоритм проверки принадлежности задаваемому ей языку, работающий полиномиальное время (т.е. число действий не превосходит полинома от длины проверяемого слова; полином может зависеть от грамматики).

Решение. Заметим, что требование полиномиальности исключает возможность решения, основанном на переборе всех возможных выводов. Тем не менее полиномиальный алгоритм существует. Поскольку практического значения он не имеет (используемые на практике КС-грамматики обладают дополнительными свойствами, позволяющими строить более эффективные алгоритмы), мы изложим лишь общую схему решения.

(1) Пусть в грамматике есть нетерминалы $K_1, \ldots, K_n$. Построим новую грамматику с нетерминалами $K'_1, \ldots, K'_n$ так, чтобы выполнялось такое свойство: из K'_i выводятся (в новой грамматике) те же слова, что из K_i в старой, за исключением пустого слова, которое не выводится.

Чтобы выполнить такое преобразование грамматики, надо выяснить, из каких нетерминалов исходной грамматики выводится пустое слово, а затем каждое правило заменить на совокупность правил, получающихся, если в правой части опустить какие-либо из нетерминалов, из которых выводится пустое слово, а у остальных поставить штрихи. Например, если в исходной грамматике было правило

$$\mathtt{K} \to \mathtt{L\ M\ N},$$

причем из `L` и `N` выводится пустое слово, а из `M` нет, то это правило надо заменить на правила

$$\mathtt{K'} \quad \to \quad \mathtt{L'M'N'}$$

$$\begin{aligned} \mathtt{K}' &\to \mathtt{M}'\mathtt{N}' \\ \mathtt{K}' &\to \mathtt{L}'\mathtt{M}' \\ \mathtt{K}' &\to \mathtt{M}' \end{aligned}$$

(2) Итак, мы свели дело к грамматике, где ни из одного нетерминала не выводится пустое слово. Теперь устраним «циклы» вида

$$\begin{aligned} \mathtt{K} &\to \mathtt{L} \\ \mathtt{L} &\to \mathtt{M} \\ \mathtt{M} &\to \mathtt{N} \\ \mathtt{N} &\to \mathtt{K} \end{aligned}$$

(в правой части каждого правила один символ, и эти символы образуют цикл произвольной длины): это легко сделать, отождествив все входящие в цикл нетерминалы.

(3) Теперь проверка принадлежности какого-либо слова языку, порождённому грамматикой, может выполняться так: для каждого подслова проверяемого слова и для каждого нетерминала выясняем, порождается ли это подслово этим нетерминалом. При этом подслова проверяются в порядке возрастания длин, а нетерминалы — в таком порядке, чтобы при наличии правила $K \to L$ нетерминал L проверялся раньше нетерминала K. (Это возможно в силу отсутствия циклов.) Поясним этот процесс на примере.

Пусть в грамматике есть правила

$$\begin{aligned} \mathtt{K} &\to \mathtt{L} \\ \mathtt{K} &\to \mathtt{M\ N\ L} \end{aligned}$$

и других правил, содержащих `K` в левой части, нет. Мы хотим узнать, выводится ли данное слово A из нетерминала `K`. Это будет так в одном из случаев:

- если A выводится из `L`;
- если A можно разбить на непустые слова B, C, D, для которых B выводится из `M`, C выводится из `N`, а D выводится из `L`.

Вся эта информация уже есть (слова B, C, D короче A, а L рассмотрен до K).

Легко видеть, что число действий этого алгоритма полиномиально. Степень полинома зависит от числа нетерминалов в правых частях правил и может быть понижена, если грамматику преобразовать к форме, в которой правая часть каждого правила не более 2 нетерминалов (это легко сделать, вводя новые нетерминалы: например, правило K $\to$ LMK можно заменить на K $\to$ LN и N $\to$ MK, где N — новый нетерминал).

13.1.6. Рассмотрим грамматику с единственным нетерминалом K, нетерминалами 1, 2, 3 и правилами

```
K → 0
K → 1 K
K → 2 K K
K → 3 K K K
```

Как проверить выводимость слова в этой грамматике, читая слово слева направо? (Число действий при прочтении одной буквы должно быть ограничено.)

Решение. Хранится целая переменная n, инвариант: слово выводимо $\Leftrightarrow$ непрочитанная часть представляет собой конкатенацию (соединение) n выводимых слов.

13.1.7. Тот же вопрос для грамматики

```
K → 0
K → K 1
K → K K 2
K → K K K 3
```

13.2. Метод рекурсивного спуска

В отличие от алгоритма предыдущего раздела (представляющего чисто теоретический интерес), алгоритмы на основе рекурсивного спуска часто используются на практике. Этот метод применим, однако, далеко не ко всем грамматикам. Мы обсудим необходимые ограничения позднее.

Идея метода рекурсивного спуска такова. Для каждого нетерминала `K` мы строим процедуру `ReadK`, которая — в применении к любому входному слову `x` — делает две вещи:

- находит наибольшее начало `z` слова `x`, которое может быть началом выводимого из `K` слова;
- сообщает, является ли найденное слово `z` выводимым из `K`.

Прежде чем описывать этот метод более подробно, договоримся о том, как процедуры получают сведения о входном слове и как сообщают о результатах своей работы. Мы предполагаем, что буквы входного слова поступают к ним по одной, т.е. имеется граница, отделяющая «прочитанную» часть от «непрочитанной». Будем считать, что есть функция (без параметров)

```
Next:  Symbol
```

дающая первый непрочитанный символ. Ее значениями могут быть терминальные символы, а также специальный символ `EOI` (End Of Input — конец входа), означающий, что всё слово уже прочитано. Вызов этой функции, разумеется, не сдвигает границы между прочитанной и непрочитанной частью — для этого есть процедура `Move`, которая сдвигает границу на один символ. (Она применима, если `Next<>EOI`.) Пусть, наконец, имеется булевская переменная `b`.

Теперь мы можем сформулировать наши требования к процедуре `ReadK`. Они состоят в следующем:

- `ReadK` прочитывает из оставшейся части слова максимальное начало A, являющееся началом некоторого слова, выводимого из `K`;
- значение `b` становится истинным или ложным в зависимости от того, является ли A выводимым из `K` или лишь невыводимым началом выводимого (из `K`) слова.

Для удобства введём такую терминологию: выводимое из `K` слово будем называть `K`-словом, а любое начало любого выводимого из `K` слова — `K`-началом. Два сформулированных требования вместе будем выражать словами «`ReadK` корректна для `K`».

Начнем с примера. Пусть правило

$$\mathtt{K} \to \mathtt{L}\ \mathtt{M}$$

является единственным правилом грамматики, содержащим K в левой части, пусть L, M — нетерминалы и ReadL, ReadM — корректные (для них) процедуры.

Рассмотрим такую процедуру:

```
procedure ReadK;
begin
| ReadL;
| if b then begin
| | ReadM;
| end;
end;
```

13.2.1. Привести пример, когда эта процедура будет некорректной для K.

Ответ. Пусть из L выводится любое слово вида 00..00, а из M выводится лишь слово 01. Тогда из K выводится слово 00001, но процедура ReadK этого не заметит.

Укажем достаточные условия корректности процедуры ReadK. Для этого нам понадобятся некоторые обозначения. Пусть фиксированы КС-грамматика и некоторый нетерминал N этой грамматики. Рассмотрим N-слово A, которое имеет собственное начало B, также являющееся N-словом (если такие есть). Для любой пары таких слов A и B рассмотрим терминальный символ, идущий в A непосредственно за B. Множество всех таких терминалов обозначим Посл(N). (Если никакое N-слово не является собственным началом другого N-слова, то множество Посл(N) пусто.)

13.2.2. Указать (а) Посл(E) для примера 1 (с. 225); (б) Посл(E) и Посл(T) для примера 2 (с. 226); (в) Посл(⟨слаг⟩) и Посл(⟨множ⟩) для примера 3 (с. 228);

Ответ. (а) Посл(E) = {[, (}. (б) Посл(E) = {[, (}; Посл(T) пусто (никакое T-слово не является началом другого). (в) Посл(⟨слаг⟩) = {*}; Посл(⟨множ⟩) пусто.

Кроме того, для каждого нетерминала N обозначим через Нач(N) множество всех терминалов, являющихся первыми буквами непустых N-слов.

Это обозначение — вместе с предыдущим — позволит дать достаточное условие корректности процедуры `ReadK` в описанной выше ситуации.

13.2.3. Доказать, что если Посл(`L`) не пересекается с Нач(`M`) и множество всех `M`-слов непусто, то `ReadK` корректна.

Решение. Рассмотрим два случая.

(1) Пусть после `ReadL` значение переменной `b` ложно. В этом случае `ReadL` читает со входа максимальное `L`-начало A, не являющееся `L`-словом. Оно является `K`-началом (здесь важно, что множество `M`-слов непусто.). Будет ли оно максимальным `K`-началом среди начал входа? Если нет, то A является началом слова BC, где B есть `L`-слово, C есть `M`-начало и BC — более длинное начало входа, чем A. Если B длиннее A, то A — не максимальное начало входа, являющееся `L`-началом, что противоречит корректности `ReadL`. Если $B = A$, то A было бы `L`-словом, а это не так. Значит, B короче A, C непусто и первый символ слова C следует в A за последним символом слова B, т.е. Посл(`L`) пересекается с Нач(`M`). Противоречие. Итак, A максимально. Из сказанного следует также, что A не является `K`-словом. Корректность процедуры `ReadK` в этом случае проверена.

(2) Пусть после `ReadL` значение переменной `b` истинно. Тогда прочитанное процедурой `ReadK` начало входа имеет вид AB, где A есть `L`-слово, а B есть `M`-начало. Тем самым AB есть `K`-начало. Проверим его максимальность. Пусть C есть большее `K`-начало. Тогда либо C есть `L`-начало (что невозможно, так как A было максимальным `L`-началом), либо $C = A'B'$, где A' — `L`-слово, B' — `M`-начало. Если A' короче A, то B' непусто и начинается с символа, принадлежащего и Нач(`M`), и Посл(`L`), что невозможно. Если A' длиннее A, то A — не максимальное `L`-начало. максимальным. Итак, $A' = A$. Но в этом случае B' есть продолжение B, что противоречит корректности `ReadM`. Итак, AB — максимальное `K`-начало. Остается проверить правильность выдаваемого процедурой `ReadK` значения переменной `b`. Если оно истинно, то это очевидно. Если оно ложно, то B не есть `M`-слово, и надо проверить, что AB — не `K`-слово. В самом деле, если бы выполнялось $AB = A'B'$, где A' — `L`-слово, B' — `M`-слово, то A' не может быть длиннее A (`ReadL` читает максимальное слово), A' не может быть равно A (тогда B' равно B и не является `M`-словом) и A' не может быть короче A (тогда первый символ B' принадлежит и Нач(`M`), и Посл(`L`)). Задача решена.

Перейдем теперь к другому частному случаю. Пусть в КС-грамматике есть правила

```
K → L
K → M
K → N
```

и других правил с левой частью K нет.

13.2.4. Считая, что ReadL, ReadM и ReadN корректны (для L, M и N) и что множества Нач(L), Нач(M) и Нач(N) не пересекаются, написать процедуру, корректную для K.

Решение. Схема процедуры такова:

```
procedure ReadK;
begin
| if (Next принадлежит Нач(L)) then begin
| | ReadL;
| end else if (Next принадлежит Нач(M)) then begin
| | ReadM;
| end else if (Next принадлежит Нач(N)) then begin
| | ReadN;
| end else begin
| | b := true или false  в зависимости от того,
| |      выводимо ли пустое слово из K или нет
| end;
end;
```

Докажем, что ReadK корректно реализует K. Если Next не принадлежит ни одному из множеств Нач(L), Нач(M), Нач(N),то пустое слово является наибольшим началом входа, являющимся K-началом. Если Next принадлежит одному (и, следовательно, только одному) из этих множеств, то максимальное начало входа, являющееся K-началом, непусто и читается соответствующей процедурой.

13.2.5. Используя сказанное, составьте процедуру распознавания выражений для грамматики (пример 3, с. 228):

$$\langle\text{выр}\rangle \quad\to\quad \langle\text{слаг}\rangle\ \langle\text{оствыр}\rangle$$

$$
\begin{aligned}
\langle\text{оствыр}\rangle &\to \texttt{+}\ \langle\text{выр}\rangle\\
\langle\text{оствыр}\rangle &\to \\
\langle\text{слаг}\rangle &\to \langle\text{множ}\rangle\ \langle\text{остслаг}\rangle\\
\langle\text{остслаг}\rangle &\to \texttt{*}\ \langle\text{слаг}\rangle\\
\langle\text{остслаг}\rangle &\to \\
\langle\text{множ}\rangle &\to \texttt{x}\\
\langle\text{множ}\rangle &\to \texttt{(}\ \langle\text{выр}\rangle\ \texttt{)}
\end{aligned}
$$

Решение. Эта грамматика не полностью подпадает под рассмотренные частные случаи: в правых частях есть комбинации терминалов и нетерминалов

$$\texttt{+}\ \langle\text{выр}\rangle$$

и группы из трех символов

$$\texttt{(}\ \langle\text{выр}\rangle\ \texttt{)}$$

В грамматике есть также несколько правил с одной левой частью и с правыми частями разного рода, например

$$
\begin{aligned}
\langle\text{оствыр}\rangle &\to \texttt{+}\ \langle\text{выр}\rangle\\
\langle\text{оствыр}\rangle &\to
\end{aligned}
$$

Эти ограничения не являются принципиальными. Так, правило типа $\texttt{K} \to \texttt{L M N}$ можно было бы заменить на два правила $\texttt{K} \to \texttt{L Q}$ и $\texttt{Q} \to \texttt{M N}$, терминальные символы в правой части — на нетерминалы (с единственным правилом замены на соответствующие терминалы). Несколько правил с одной левой частью и разнородными правыми также можно свести к уже разобранному случаю: например,

$$
\begin{aligned}
\texttt{K} &\to \texttt{L M N}\\
\texttt{K} &\to \texttt{P Q}\\
\texttt{K} &\to
\end{aligned}
$$

можно заменить на правила

$$
\begin{aligned}
\mathtt{K} &\to \mathtt{K}_1\\
\mathtt{K} &\to \mathtt{K}_2\\
\mathtt{K} &\to \mathtt{K}_3\\
\mathtt{K}_1 &\to \mathtt{L\ M\ N}\\
\mathtt{K}_2 &\to \mathtt{P\ Q}\\
\mathtt{K}_3 &\to
\end{aligned}
$$

Но мы не будем этого делать — а сразу же запишем то, что получится, если подставить описания процедур для новых терминальных символов в места их использования. Например, для правила

$$\mathtt{K} \to \mathtt{L\ M\ N}$$

это даёт процедуру

```
procedure ReadK;
begin
| ReadL;
| if b then begin ReadM; end;
| if b then begin ReadN; end;
end;
```

Для её корректности надо, чтобы Посл(L) не пересекалось с Нач(MN) (которое равно Нач(M), если из M не выводится пустое слово, и равно объединению Нач(M) и Нач(N), если выводится), а также чтобы Посл(M) не пересекалось с Нач(N).

Аналогичным образом правила

$$
\begin{aligned}
\mathtt{K} &\to \mathtt{L\ M\ N}\\
\mathtt{K} &\to \mathtt{P\ Q}\\
\mathtt{K} &\to
\end{aligned}
$$

приводят к процедуре

```
procedure ReadK;
begin
  if (Next принадлежит Нач(LMN)) then begin
    ReadL;
    if b then begin ReadM; end;
    if b then begin ReadN; end;
  end else if (Next принадлежит Нач(PQ)) then begin
    ReadP;
    if b then begin ReadQ; end;
  end else begin
    b := true;
  end;
end;
```

корректность которой требует, чтобы Нач(LMN) не пересекалось с Нач(PQ).

Читая приведенную далее программу, полезно иметь в виду соответствие между русскими и английскими словами:

ВЫРажение	EXPRession
ОСТаток ВЫРажения	REST of EXPRession
СЛАГаемое	ADDitive term
ОСТаток СЛАГаемого	REST of ADDitive term
МНОЖитель	FACTor

```
procedure ReadSymb (c: Symbol);
  b := (Next = c);
  if b then begin Move; end;
end;

procedure ReadExpr;
  ReadAdd;
  if b then begin ReadRestExpr; end;
end;

procedure ReadRestExpr;
  if Next = '+' then begin
```

```
    ReadSymb ('+');
    if b then begin ReadExpr; end;
  end else begin
    b := true;
  end;
end;

procedure ReadAdd;
  ReadFact;
  if b then begin ReadRestAdd; end;
end;

procedure ReadRestAdd;
  if Next = '*' then begin
    ReadSymb ('*');
    if b then begin ReadAdd; end;
  end else begin
    b := true;
  end;
end;

procedure ReadFact;
  if Next = 'x' then begin
    ReadSymb ('x');
  end else if Next = '(' then begin
    ReadSymb ('(');
    if b then begin ReadExpr; end;
    if b then begin ReadSymb (')'); end;
  end else begin
    b := false;
  end;
end;
```

Осталось обсудить проблемы, связанные с взаимной рекурсивностью этих процедур (одна использует другую и наоборот). В паскале это допускается, только требуется дать предварительное описание процедур («forward»).

Как всегда для рекурсивных процедур, помимо доказательства того, что каждая процедура работает правильно в предположении, что используемые в ней вызовы процедур работают правильно, надо доказать отдельно, что работа завершается. (Это не очевидно: если в грамматике есть правило K $\to$ KK, то из K ничего не выводится, Посл(K) и Нач(K) пусты, но написанная по нашим канонам процедура

```
procedure ReadK;
begin
| ReadK;
| if b then begin
| | ReadK;
| end;
end;
```

не заканчивает работы.)

В даннном случае процедуры ReadRestExpr, ReadRestAdd, ReadFact либо завершаются, либо уменьшают длину непрочитанной части входа. Поскольку любой цикл вызовов включает одну из них, то зацикливание невозможно. Задача решена.

13.2.6. Пусть в грамматике имеются два правила с нетерминалом K в левой части, имеющих вид

$$\begin{aligned} K &\to L\ K \\ K &\to \end{aligned}$$

по которым K-слово представляет собой конечную последовательность L-слов, причем множества Посл(L) и Нач(K) (в данном случае равное Нач(L)) не пересекаются. Используя корректную для L процедуру ReadL, написать корректную для K процедуру ReadK, не используя рекурсии.

Решение. По нашим правилам следовало бы написать

```
procedure ReadK;
begin
| if (Next принадлежит Нач(L)) then begin
| | ReadL;
```

```
    | if b then begin ReadK; end;
  end else begin
    b := true;
  end;
end;
```

завершение работы гарантируется тем, что перед рекурсивным вызовом длина непрочитанной части уменьшается.

Эта рекурсивная процедура эквивалентна нерекурсивной:

```
procedure ReadK;
begin
  b := true;
  while b and (Next принадлежит Нач(L)) do begin
    ReadL;
  end;
end;
```

Формально можно проверить эту эквивалентность так. Завершаемость в обоих случаях ясна. Достаточно проверить поэтому, что тело рекурсивной процедуры эквивалентно нерекурсивной в предположении, что её рекурсивный вызов эквивалентен вызову нерекурсивной процедуры. Подставим:

```
if (Next принадлежит Нач(L)) then begin
  ReadL;
  if b then begin
    b := true;
    while b and (Next принадлежит Нач(L)) do begin
      ReadL;
    end;
  end;
end else begin
  b := true;
end;
```

Первую команду `b:=true` можно выкинуть (в этом месте и так b истинно). Вторую команду можно перенести в начало:

```
b := true;
if (Next принадлежит Нач(L) then begin
| ReadL;
| if b then begin
| | while b and (Next принадлежит Нач(L)) do begin
| | | ReadL;
| | end;
| end;
end;
```

Теперь внутренний `if` можно выкинуть (если b ложно, цикл `while` всё равно не выполняется) и добавить в условие внешнего `if` условие b (которое всё равно истинно).

```
b := true;
if b and (Next принадлежит Нач(L)) then begin
| ReadL;
| while b and (Next принадлежит Нач(L)) do begin
| | ReadL;
| end;
end;
```

что эквивалентно приведенной выше нерекурсивной процедуре (из которой вынесена первая итерация цикла).

13.2.7. Доказать корректность приведенной выше нерекурсивной программы непосредственно, без ссылок на рекурсивную.

Решение. Рассмотрим наибольшее начало входа, являющееся K-началом. Оно представляется в виде конкатенации (последовательного приписывания) нескольких непустых L-слов и, возможно, одного непустого L-начала, не являющегося L-словом. Инвариант цикла: прочитано несколько из них; b $\Leftrightarrow$ (последнее прочитанное является L-словом).

Сохранение инварианта: если осталось последнее слово, это очевидно; если осталось несколько, то за первым L-словом (из числа оставшихся) идет символ из Нач(L), и потому это слово — максимальное начало входа, являющееся L-началом.

На практике при записи грамматики используют сокращения. Если правила для какого-то нетерминала K имеют вид

$$
\begin{aligned}
\mathtt{K} &\to \mathtt{L\ K} \\
\mathtt{K} &\to
\end{aligned}
$$

(т.е. K-слова — это последовательности L-слов), то этих правил не пишут, а вместо K пишут L в фигурных скобках. Несколько правил с одной левой частью и разными правыми записывают как одно правило, разделяя альтернативные правые части вертикальной чертой.

Например, рассмотренная выше грамматика для ⟨выр⟩ может быть записана так:

$$
\begin{aligned}
\langle\text{выр}\rangle &\to \langle\text{слаг}\rangle\ \{\ \mathtt{+}\ \langle\text{слаг}\rangle\ \} \\
\langle\text{слаг}\rangle &\to \langle\text{множ}\rangle\ \{\ \mathtt{*}\ \langle\text{множ}\rangle\ \} \\
\langle\text{множ}\rangle &\to \mathtt{x} \mid (\ \langle\text{выр}\rangle\ )
\end{aligned}
$$

13.2.8. Написать процедуру, корректную для ⟨выр⟩, следуя этой грамматике и используя цикл вместо рекурсии, где можно.

Решение.

```
procedure ReadSymb (c: Symbol);
| b := (Next = c);
| if b then begin Move; end;
end;

procedure ReadExpr;
begin
| ReadAdd;
| while b and (Next = '+') do begin
| | Move; ReadAdd;
| end;
end;

procedure ReadAdd;
```

```
begin
| ReadFact;
| while b and (Next = '*') do begin
| | Move; ReadFact;
| end;
end;

procedure ReadFact;
begin
| if Next = 'x' do begin
| | Move; b := true;
| end else if Next = '(' then begin
| | Move; ReadExpr;
| | if b then begin ReadSymb (')'); end;
| end else begin
| | b := false;
| end;
end;
```

13.2.9. В последней процедуре команду `b:=true` можно опустить. Почему?

Решение. Можно предполагать, что все процедуры вызываются при `b=true`.

13.3. Алгоритм разбора для LL(1)-грамматик

В этом разделе мы рассмотрим ещё один метод проверки выводимости в КС-грамматике, называемый по традиции LL(1)-разбором. Вот его идея в одной фразе: можно считать, что в процессе вывода мы всегда заменяем самый левый нетерминал и нужно лишь выбрать одно из правил; если нам повезёт с грамматикой, то выбрать правило можно, глядя на первый символ выводимого из этого нетерминала слова. Говоря более формально, дадим такое

Определение. *Левым выводом* (слова в грамматике) называется вывод, в котором на каждом шаге замене подвергается самый левый из нетерминалов.

13.3.1. Для каждого выводимого слова (из терминалов) существует его левый вывод.

Решение. Различные нетерминалы заменяются независимо; если в процессе вывода появилось слово $\ldots K \ldots L \ldots$, где K, L — нетерминалы, то замены K и L можно производить в любом порядке. Поэтому можно перестроить вывод так, чтобы стоящий левее нетерминал заменялся раньше. (Формально говоря, надо доказывать индукцией по длине вывода такой факт: если из некоторого нетерминала K выводится некоторое слово A, то существует левый вывод A из K.)

13.3.2. В грамматике с 4 правилами

$$\begin{array}{ll} (1) & \texttt{E} \to \\ (2) & \texttt{E} \to \texttt{TE} \\ (3) & \texttt{T} \to \texttt{(E)} \\ (4) & \texttt{T} \to \texttt{[E]} \end{array}$$

найти левый вывод слова $A = \texttt{[()([])]}$ и доказать, что он единствен.

Решение. На первом шаге можно применить только правило (2):

$$\texttt{E} \to \texttt{TE}$$

Что будет дальше с `T`? Так как слово A начинается на `[`, то может примениться только правило (4):

$$\texttt{E} \to \texttt{TE} \to \texttt{[E]E}$$

Первое `E` должно замениться на `TE` (иначе вторым символом была бы скобка `]`):

$$\texttt{E} \to \texttt{TE} \to \texttt{[E]E} \to \texttt{[TE]E}$$

и `T` должно заменяться по (3):

$$\texttt{E} \to \texttt{TE} \to \texttt{[E]E} \to \texttt{[TE]E} \to \texttt{[(E)E]E}$$

Далее первое `E` должно замениться на пустое слово (иначе третьей буквой слова будет `(` или `[` — только на эти символы может начинаться слово, выводимое из `T`):

$$\texttt{E} \to \texttt{TE} \to \texttt{[E]E} \to \texttt{[TE]E} \to \texttt{[(E)E]E} \to \texttt{[()E]E}$$

и далее

$$\ldots \to \texttt{[()TE]E} \to \texttt{[()(E)E]E} \to \texttt{[()(TE)E]E} \to \texttt{[()([E]E)E]E} \to$$

$$\to \texttt{[()([]E)E]E} \to \texttt{[()([])E]E} \to \texttt{[()([])]E} \to \texttt{[()([])]}$$

Что требуется от грамматики, чтобы такой метод поиска левого вывода был применим? Пусть, например, на очередном шаге самым левым нетерминалом оказался нетерминал K, т.е. мы имеем слово вида AKU, где A — слово из терминалов, а U — слово из терминалов и нетерминалов. Пусть в грамматике есть правила

$$\begin{aligned} K &\to LMN \\ K &\to PQ \\ K &\to R \end{aligned}$$

Нам надо выбрать одно из них. Мы будем пытаться сделать этот выбор, глядя на первый символ той части входного слова, которая выводится из KU.

Рассмотрим множество Нач(LMN) тех терминалов, с которых начинаются непустые слова, выводимые из LMN. (Это множество равно Нач(L), объединенному с Нач(M), если из L выводится пустое слово, а также с Нач(N), если из L и из M выводится пустое слово.) Чтобы описанный метод был применим, надо, чтобы Нач(LMN), Нач(PQ) и Нач(R) не пересекались. Но этого мало. Ведь может быть так, например, что из LMN будет выведено пустое слово, а из слова U будет выведено слово, начинающееся на букву из Нач(PQ). Следующие определения учитывают эту проблему.

Напомним, что определение выводимости в КС-грамматике было дано только для слова из терминалов. Оно очевидным образом обобщается на случай слов из терминалов и нетерминалов. Можно также говорить о выводимости одного слова (содержащего терминалы и нетерминалы) из другого. (Если говорится о выводимости слова без указания того, откуда оно выводится, то всегда подразумевается выводимость в грамматике, т.е. выводимость из начального нетерминала.)

Для каждого слова X из терминалов и нетерминалов через Нач(X) обозначаем множество всех терминалов, с которых начинаются непустые слова из терминалов, выводимые из X. (В случае, если из любого нетерминала выводится хоть одно слово из терминалов, не играет роли, рассматриваем ли мы при определении Нач(X) слова только из терминалов или любые слова. Мы будем предполагать далее, что это условие выполнено.)

Для каждого нетерминала K через Послед(K) обозначим множество терминалов, которые встречаются в выводимых (в грамматике) словах сразу же за K. (Не смешивать с Посл(K) предыдущего раздела!) Кроме того, в Послед(K) включается символ `EOI`, если существует выводимое слово, оканчивающееся на K.

Для каждого правила

$$\mathrm{K} \to V$$

(где K — нетерминал, V — слово, содержащее терминалы и нетерминалы) определим множество *направляющих терминалов*, обозначаемое Напр($\mathrm{K} \to V$). По определению оно равно Нач(V), к которому добавлено Послед(K), если из V выводится пустое слово.

Определение. Грамматика называется LL(1)-грамматикой, если для любых правил $\mathrm{K} \to V$ и $\mathrm{K} \to W$ с одинаковыми левыми частями множества Напр($\mathrm{K} \to V$) и Напр($\mathrm{K} \to W$) не пересекаются.

13.3.3. Является ли грамматика

```
K  →  K #
K  →
```

(выводимыми словами являются последовательности диезов) LL(1)-грамматикой?

Решение. Нет: символ # принадлежит множествам направляющих символов для обоих правил (для второго — поскольку # принадлежит Послед(K)).

13.3.4. Написать LL(1)-грамматику для того же языка.

Решение.

```
K  →  # K
K  →
```

Как говорят, «леворекурсивное» правило заменено на «праворекурсивное».

Следующая задача показывает, что для LL(1)-грамматики существует не более одного возможного продолжения левого вывода.

13.3.5. Пусть дано выводимое в LL(1)-грамматике слово X, в котором выделен самый левый нетерминал K: $X = AKS$, где A — слово из терминалов, S — слово из терминалов и нетерминалов. Пусть существуют два различных правила грамматики с нетерминалом K в левой части, и мы применили их к выделенному в X нетерминалу K, затем продолжили вывод и в конце концов получили два слова из терминалов, начинающихся на A. Доказать, что в этих словах за началом A идут разные буквы. (Здесь к числу букв мы относим `EOI`.)

Решение. Эти буквы принадлежат направляющим множествам различных правил.

13.3.6. Доказать, что если слово выводимо в LL(1)-грамматике, то его левый вывод единствен.

Решение. Предыдущая задача показывает, что на каждом шаге левый вывод продолжается однозначно.

13.3.7. Грамматика называется *леворекурсивной*, если из некоторого нетерминала K выводится слово, начинающееся с K, но не совпадающее с ним. Доказать, что леворекурсивная грамматика, в которой из каждого нетерминала выводится хотя бы одно непустое слово из терминалов и для каждого нетерминала существует вывод (начинающийся с начального нетерминала), в котором он встречается, не является LL(1)-грамматикой.

Решение. Пусть из K выводится KU, где K — нетерминал, а U — непустое слово. Можно считать, что это левый вывод (другие нетерминалы можно не заменять). Рассмотрим вывод

$$K \rightsquigarrow KU \rightsquigarrow KUU \rightsquigarrow \ldots$$

(знак $\rightsquigarrow$ обозначает несколько шагов вывода) и левый вывод $K \rightsquigarrow A$, где A — непустое слово из терминалов. На каком-то шаге второй вывод отклоняется от первого, а между тем по обоим путям может быть получено слово, начинающееся на A (в первом случае это возможно, так как сохраняется нетерминал K, который может впоследствии быть заменён на A). Это противоречит возможности однозначного определения правила, применяемого на очередном шаге поиска левого вывода. (Однозначность выполняется для выводов из начального нетерминала, и надо воспользоваться тем, что K по предположению встречается в таком выводе.)

Таким образом, к леворекурсивным грамматикам (кроме тривиальных случаев) LL(1)-метод неприменим. Их приходится преобразовывать к эквивалентным LL(1)-грамматикам — или пользоваться другими методами распознавания.

13.3.8. Используя сказанное, построить алгоритм проверки выводимости слова из терминалов в LL(1)-грамматике.

Решение. Мы следуем описанному выше методу поиска левого вывода, храня лишь часть слова, находящуюся правее уже прочитанной части входного слова. Другими словами, мы храним слово `S` из терминалов и нетерминалов, обладающее такими свойствами (прочитанную часть входа обозначаем через `A`):

1) слово `AS` выводимо в грамматике;

2) любой левый вывод входного слова проходит через стадию `AS`

Эти свойства вместе будем обозначать «(И)».

Вначале `A` пусто, а `S` состоит из единственного символа — начального нетерминала.

Если в некоторый момент `S` начинается на терминал `t` и `t` = `Next`, то можно выполнить команду `Move` и удалить символ `t`, являющийся начальным в `S`, поскольку при этом `AS` не меняется.

Если `S` начинается на терминал `t` и `t` $\neq$ `Next`, то входное слово невыводимо — ибо по условию любой его вывод должен проходить через `AS`. (Это же справедливо и в случае `Next` = `EOI`.)

Если `S` пусто, то из условия (И) следует, что входное слово выводимо тогда и только тогда, когда `Next` = `EOI`.

Остается случай, когда `S` начинается с некоторого нетерминала `K`. По доказанному выше все левые выводы из `S` слов, начинающихся на символ `Next`, начинаются с применения к `S` одного и того же правила — того, для которого `Next` принадлежит направляющему множеству. Если таких правил нет, то входное слово невыводимо. Если такое правило есть, то нужно применить его к первому символу слова S — при этом свойство (И) не нарушится. Приходим к такому алгоритму:

```
S := пустое слово;
error := false;
{error => входное слово невыводимо;}
{not error => (И)}
while (not error) and not ((Next=EOI) and (S пусто))
    do begin
  if (S начинается на терминал, равный Next) then begin
    Move; удалить из S первый символ;
  end else if (S начинается на терминал, не равный Next)
            then begin
    error := true;
  end else if (S пусто) and (Next <> EOI) then begin
    error := true;
  end else if (S начинается на нетерминал и Next входит в
      направляющее множество одного из правил для этого
      нетерминала) then begin
    применить это правило
  end else if (S начинается на нетерминал и Next не входит
    в направляющее множество ни одного из правил для этого
    нетерминала) then begin
    error := true;
  end else begin
    {так не бывает}
  end;
end;
{входное слово выводимо <=> not error}
```

Алгоритм заканчивает работу, поскольку при появлении терминала в начале слова S происходит чтение со входа или остановка, а бесконечный цикл сменяющих друг друга нетерминалов в начале S означал бы, что грамматика леворекурсивна. (А мы можем предполагать, согласно предыдущей задаче, что это не так: нетерминалы, не встречающиеся в выводах, а также