```
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, a 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.

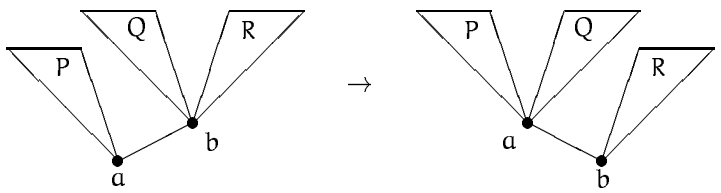
Решение. Максимальное число вершин равно $2^n - 1$. Если m_n — минимальное число вершин, то, как легко видеть, $m_{n+2} = 1 + m_n + m_{n+1}$, откуда $m_n = \Phi_{n+1} - 1$ (Φ_n — n-ое число Фибоначчи, $\Phi_0 = 1$, $\Phi_1 = 1$, $\Phi_{n+2} = \Phi_n + \Phi_{n+1}$).

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

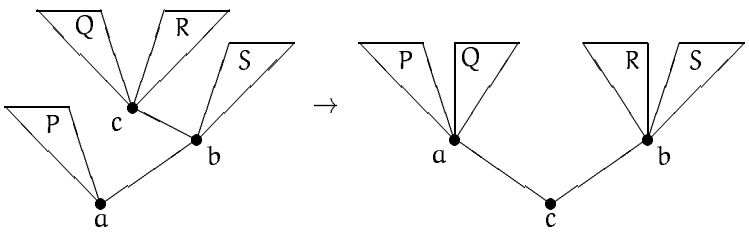
Вращения

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 $P < a < Q < c < R < b < S$.

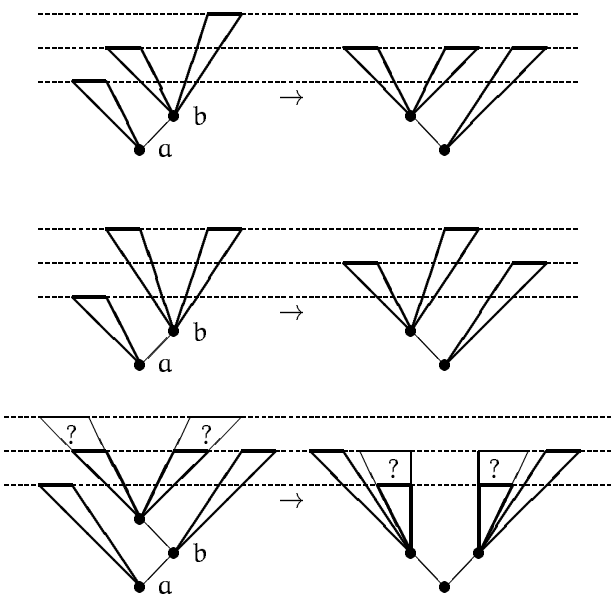


Такой же порядок соответствует дереву с корнем c, имеющим левого сына a и правого сына b, для которого P и Q — поддеревья вершины a, а R и S — поддеревья вершины b. Соответствующее преобразование будем называть *большим правым вращением*. (Аналогично определяется симметричное ему *большое левое вращение*.)

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:

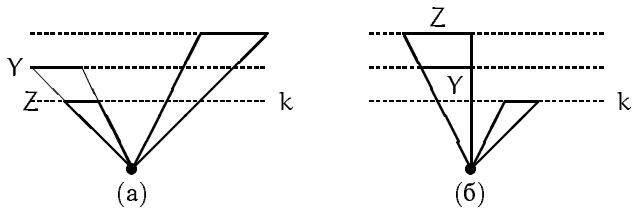
Лемма. Если в сбалансированном дереве X одно из его поддеревьев Y заменили на сбалансированное дерево Z, причем высота Z отличается от высоты Y не более чем на 1, то полученное такой «прививкой» дерево можно превратить в сбалансированное вращениями (причем количество вращений не превосходит высоты, на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

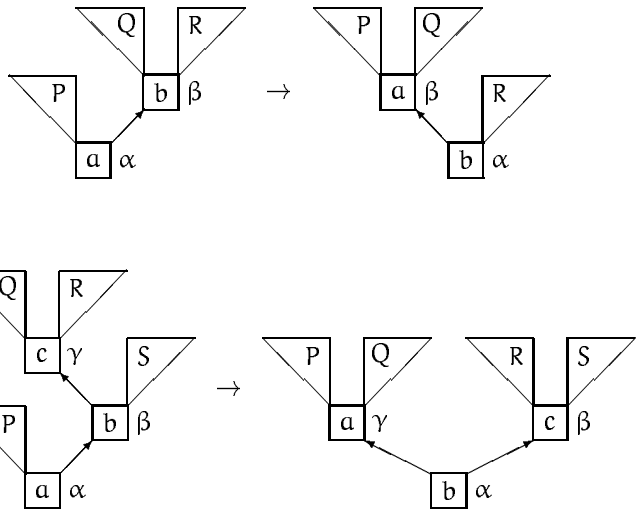
12.2.5. Составить программы добавления и удаления элементов, сохраняющие сбалансированность. Число действий не должно превосходить $C \cdot$(высота дерева). Разрешается хранить в вершинах дерева дополнительную информацию, необходимую при балансировке.

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

```
diff [i]  =  (высота правого поддерева вершины i) −
                (высота левого поддерева вершины i).
```

Нам потребуются четыре процедуры, соответствующие большим и малым правым и левым вращениями. 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, ... 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 висходном 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. 12.2) 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 неполным, 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 [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...BCD...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:

⟨выр⟩ → ⟨слаг⟩ ⟨оствыр⟩
⟨оствыр⟩ → + ⟨выр⟩
⟨оствыр⟩ →
⟨слаг⟩ → ⟨множ⟩ ⟨остслаг⟩
⟨остслаг⟩ → * ⟨слаг⟩
⟨остслаг⟩ →
⟨множ⟩ → x
⟨множ⟩ → (⟨выр⟩)

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:

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

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

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

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

$$\texttt{K}' \to \texttt{L}'\texttt{M}'\texttt{N}'$$

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

(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 $\mathtt{K} \to \mathtt{L}$ нетерминал L проверялся раньше нетерминала K. (Это возможно в силу отсутствия циклов.) Поясним этот процесс на примере.

Пусть в грамматике есть правила

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

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 содержит 1 или 2 нетерминала (это легко сделать, вводя новые нетерминалы: например, правило K → LMK можно заменить на K → LN и N → 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 ⇔ 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 KC-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. 220); (б) Посл(E) и Посл(T) для примера 2 (с. 221); (в) Посл(⟨слаг⟩) и Посл(⟨множ⟩) для примера 3 (с. 223);

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

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

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. 223):

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

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

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

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

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 едиственным правилом замены на соответствующие терминалы). Несколько правил с одной левой частью и разнородными правыми также можно свести к уже разобранному случаю: например,

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

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 → 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 данном 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 нетермиинаном 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. Предполагается, что пустое слово не выводимо из L.

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 пустое слово не выводимо из L (и, следовательно, 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

$$\begin{aligned} &(1)\quad \mathtt{E} \to \\ &(2)\quad \mathtt{E} \to \mathtt{TE} \\ &(3)\quad \mathtt{T} \to \mathtt{(E)} \\ &(4)\quad \mathtt{T} \to \mathtt{[E]} \end{aligned}$$

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

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

$$\mathtt{E} \to \mathtt{TE}$$

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

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

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

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

и T должно заменяться по (3):

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

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

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

и далее

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

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

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

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

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

$$K \to V$$

(где K — нетерминал, V — слово, содержащее терминалы и нетерминалы) определим множество *направляющих терминалов*, обозначаемое Напр($K \to V$). 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 $K \to V$ и $K \to W$ с одинаковыми левыми частями множества Напр($K \to V$) и Напр($K \to W$) 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.

$$\mathtt{K} \;\to\; \mathtt{\#\ K}$$
$$\mathtt{K} \;\to$$

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 нетерминалы, из которых не выводится непустого слова, несложно удалить из грамматики.)

Замечания.

- Приведенный алгоритм использует S как стек (все действия производятся с левого конца).
- Действия двух последних вариантов внутри цикла не приводят к чтению очередного символа со входа, поэтому их можно заранее предвычислить для каждого нетерминала и каждого символа `Next`. После этого на каждом шаге цикла будет читаться очередной символ входа.
- При практической реализации удобно составить таблицу, в которой записаны варианты действий в зависимости от входного символа и первого символа S, и небольшую программу, выполняющую действия в соответствии с этой таблицей.

13.3.9. При проверке того, относится ли данная грамматика к типу LL(1), необходимо вычислить Послед(T) и Нач(T) для всех нетерминалов T. Как это сделать?

Решение. Если в грамматике есть, например, правило $K \to L\,M\,N$, то

$$
\begin{array}{rcll}
\text{Нач}(L) & \subset & \text{Нач}(K), & \\
\text{Нач}(M) & \subset & \text{Нач}(K), & \text{если из } L \text{ выводимо пустое слово,} \\
\text{Нач}(N) & \subset & \text{Нач}(K), & \text{если из } L \text{ и } M \text{ выводимо пустое слово,} \\
\text{Послед}(K) & \subset & \text{Послед}(N), & \\
\text{Послед}(K) & \subset & \text{Послед}(M), & \text{если из } N \text{ выводимо пустое слово,} \\
\text{Послед}(K) & \subset & \text{Послед}(L), & \text{если из } M \text{ и } N \text{ выводимо пустое слово,} \\
\text{Нач}(N) & \subset & \text{Послед}(M), & \\
\text{Нач}(M) & \subset & \text{Послед}(L), & \\
\text{Нач}(N) & \subset & \text{Послед}(L), & \text{если из } M \text{ выводимо пустое слово,}
\end{array}
$$

Подобные правила позволяют постепенно порождать множества Нач(T), а затем и Послед(T), для всех терминалов и нетерминалов T. При этом

началом служит

$$\texttt{EOI} \in \text{Послед}(\mathsf{K})$$

для начального нетерминала K и

$$z \in \text{Нач}(z)$$

для любого терминала z. Порождение заканчивается, когда применение правил перестаёт давать новые элементы множеств Нач(T) и Послед(T).

14. Синтаксический разбор слева направо (LR)

Сейчас мы рассмотрим еще один метод синтаксического разбора, называемый LR(1)-разбором, а также некоторые упрощённые его варианты.

14.1. LR-процессы

Два отличия LR(1)-разбора от LL(1)-разбора: во-первых, строится не левый вывод, а правый, во-вторых, он строится не с начала, а с конца. (Вывод в КС-грамматике называется *правым*, если на каждом шаге замене подвергается самый правый нетерминал.)

14.1.1. Доказать, что если слово, состоящее из терминалов, выводимо, то оно имеет правый вывод.

Нам будет удобно смотреть на правый вывод «задом наперёд». Определим понятие LR-*процесса над словом* A. В этом процессе, помимо A, будет участвовать и другое слово S, которое может содержать как терминалы, так и нетерминалы. Вначале слово S пусто. В ходе LR-процесса разрешены два вида действий:

(1) можно перенести первый символ слова A (его называют *очередным* символом и обозначают `Next`) в конец слова S, удалив его из A (это действие называют *сдвигом*);

(2) если правая часть одного из правил грамматики оказалась концом слова S, то разрешается заменить её на нетерминал, стоящий в левой части этого правила; при этом слово A не меняется. (Это действие называют *свёрткой*, или *приведением*.

Отметим, что LR-процесс не является детерминированным: в одной и той же ситуации могут быть разрешены разные действия.

Говорят, что LR-процесс на слове A *успешно завершается*, если слово A становится пустым, а в слове S остается единственный нетерминал — начальный нетерминал грамматики.

14.1.2. Доказать, что для любого слова A (из терминалов) успешно завершающийся LR-процесс существует тогда и только тогда, когда слово A выводимо в грамматике. В ходе доказательства установить взаимно

однозначное соответствие между правыми выводами и успешно завершающимися LR-процессами.

Решение. При сдвиге слово SA не меняется, при свёртке слово SA подвергается преобразованию, обратному шагу вывода. Этот вывод будет правым, так как сворачивается конец S, а в A все символы — терминальные. Таким образом, каждому LR-процессу соответствует правый вывод. Обратное соответствие: пусть дан правый вывод. Представим себе, что за последним нетерминалом в слове стоит перегородка. Применив к этому нетерминалу правило грамматики, мы должны сдвинуть перегородку влево (если правая часть правила кончается на терминал). Разбивая этот сдвиг на отдельные шаги, получим процесс, в точности обратный LR-процессу.

Поскольку в ходе LR-процесса все изменения в слове S происходят с правого конца, слово S называют *стеком* LR-процесса.

Задача построения правого вывода для данного слова сводится, таким образом, к правильному выбору очередного шага LR-процесса. Нам нужно решить, будем ли мы делать сдвиг или свёртку, и если свёртку, то по какому правилу — ведь подходящих правил может быть несколько. В LR(1)-алгоритме это решение принимается на основе S и первого символа слова A; если используется только S, то говорят о LR(0)-алгоритме. (Точные определения смотри ниже.)

Пусть фиксирована грамматика, в которой из любого нетерминала можно вывести какое-либо слово из терминалов. (Это ограничение мы будет всегда предполагать выполненным.)

Пусть $\mathtt{K} \to \mathtt{U}$ — одно из правил грамматики ($\mathtt{K}$ — нетерминал, $\mathtt{U}$ — слово из терминалов и нетерминалов). Определим множество слов (из терминалов и нетерминалов), называемое *левым контекстом* правила $\mathtt{K} \to \mathtt{U}$. (Обозначение: ЛевКонт($\mathtt{K} \to \mathtt{U}$).) По определению в него входят все слова, которые являются содержимым стека непосредственно перед свёрткой $\mathtt{U}$ в $\mathtt{K}$ в ходе некоторого успешно завершающегося LR-процесса.

14.1.3. Переформулировать это определение на языке правых выводов.

Решение. Рассмотрим все правые выводы вида

$$\langle\text{начальный нетерминал}\rangle \rightsquigarrow \mathtt{XKA} \to \mathtt{XUA},$$

где A — слово из терминалов, X — слово из терминалов и нетерминалов. Все возникающие при этом слова XU и образуют левый контекст правила K → U. Чтобы убедиться в этом, следует вспомнить, что мы предполагаем, что из любого нетерминала можно вывести какое-то слово из терминалов, так что правый вывод слова XUA может быть продолжен до правого вывода какого-то слова из терминалов.

14.1.4. Все слова из ЛевКонт(K → U) кончаются, очевидно, на U. Доказать, что если у всех них этот конец U отбросить, то полученное множество слов не зависит от того, какое из правил для нетерминала K выбрано. (Это множество обозначается Лев(K).)

Решение. Из предыдущей задачи ясно, что Лев(K) — это всё, что может появиться в правых выводах левее самого правого нетерминала K.

14.1.5. Доказать, что в предыдущей фразе можно отбросить слова «самого правого»: Лев(K) — это всё то, что может появляться в правых выводах левее любого вхождения нетерминала K.

Решение. Продолжив построение правого вывода, все нетерминалы справа от K можно заменить на терминалы (а слева от K при этом ничего не изменится).

14.1.6. Построить грамматику, содержащую для каждого нетерминала K исходной грамматики нетерминал ⟨ЛевK⟩, причем следующее свойство должно выполняться для любого нетерминала K исходной грамматики: в новой грамматике из ⟨ЛевK⟩ выводимы все элементы Лев(K) и только они.

Решение. Пусть P — начальный нетерминал грамматики. Тогда в новой грамматике будет правило

$$\langle\text{ЛевP}\rangle \to \qquad \text{(пустое слово)}$$

Для каждого правила исходной грамматики, например, правила

$$\text{K} \to \text{L t M N} \qquad (\text{L, M, N — нетерминалы, t — терминал}),$$

в новую грамматику мы добавим правила

$$\begin{aligned}
\langle\text{ЛевL}\rangle &\to \langle\text{ЛевK}\rangle \\
\langle\text{ЛевM}\rangle &\to \langle\text{ЛевK}\rangle\,\text{L t} \\
\langle\text{ЛевN}\rangle &\to \langle\text{ЛевK}\rangle\,\text{L t M}
\end{aligned}$$

и аналогично поступим с другими правилами. Смысл новых правил таков: пустое слово может появиться слева от P; если слово X может появиться слева от K, то X может появиться слева от L, XLt может появиться слева от M, XLtM — слева от N. Индукцией по длине правого вывода легко проверить, что все, что может появиться слева от какого-то нетерминала, появляется в соответствии с этими правилами.

14.1.7. Почему в предыдущей задаче важно, что мы рассматриваем только правые выводы?

Ответ. В противном случае следовало бы учитывать преобразования, происходящие внутри слова, стоящего слева от K.

14.1.8. Для данной грамматики построить алгоритм, который по любому слову выясняет, каким из множеств Лев(K) оно принадлежит.

(Замечание для знатоков. Существование такого алгоритма — и даже конечного автомата, то есть индуктивного расширения с конечным числом значений, см. 1.3, — вытекает из предыдущей задачи, так как построенная в ней грамматика имеет специальный вид: в правых частях всего один нетерминал, причем он стоит у левого края. Тем не менее мы приведём явное построение.)

Решение. Будем называть ситуацией данной грамматики одно из её правил, в правой части которого отмечена одна из позиций (до первой буквы, между первой и второй буквой, ..., после последней буквы). Например, правило

$$K \to LtMN$$

(K, L, M, N — нетерминалы, t — терминал) порождает пять ситуаций

$$K \to _LtMN \quad K \to L_tMN \quad K \to Lt_MN \quad K \to LtM_N \quad K \to LtMN_$$

(позиция указывается знаком подчеркивания).

Будем говорить, что слово S *согласовано* с ситуацией $K \to U_V$, если S кончается на U, то есть $S = TU$ при некотором T, и, кроме того, T принадлежит Лев(K). (Смысл этого определения примерно таков: в стеке S подготовлена часть U для будущей свёртки UV в K.) В этих терминах ЛевКонт($K \to X$) — это множество всех слов, согласованных с ситуацией $K \to X_$, а Лев(K) — это множество всех слов, согласованных с ситуацией $K \to _X$ (где $K \to _X$) — любое правило для нетерминала K).

Эквивалентное определение в терминах LR-процесса: S согласовано с ситуацией K → U_V, если существует успешный LR-процесс, в котором события развиваются так:

- в ходе процесса в стеке появляется слово S, и оно оканчивается на U;
- некоторое время S не затрагивается, а справа от него появляется V;
- UV сворачивается в K;
- процесс продолжается и успешно завершается.

14.1.9. Доказать эквивалентность этих определений.

[Указание. Если S = TU и T принадлежит Лев(K), то можно получить в стеке сначала T, потом U, потом V, потом свернуть UV в K и затем успешно завершить процесс. (Мы используем несколько раз тот факт, что из любого нетерминала что-то да выводится: благодаря этому мы можем добавить в стек любое слово.)]

Наша цель — построение алгоритма, распознающего принадлежность произвольного слова к Лев(K). Рассмотрим функцию, сопоставляющую с каждым словом S (из терминалов и нетерминалов) множество всех согласованных с ним ситуаций. Это множество назовём *состоянием, соответствующим слову S*. Будем обозначать его Сост(S). Достаточно показать, что функция Сост(S) индуктивна, то есть что значение Сост(SJ), где J — терминал или нетерминал, может быть вычислено, если известно Сост(S) и символ J. (Мы видели ранее, как принадлежность к Лев(K) выражается в терминах этой функции.) Значение Сост(SJ) вычисляется по таким правилам:

> (1) Если слово S согласовано с ситуацией K → U_V, причём слово V начинается на букву J, то есть V = JW, то SJ согласовано с ситуацией K → UJ_W.

Это правило полностью определяет все ситуации с непустой левой половиной (то есть не начинающиеся с подчёркивания), согласованные с SJ. Осталось определить, для каких нетерминалов K слово SJ принадлежит Лев(K). Это делается по двум правилам:

(2) Если ситуация L → U _ V согласована с SJ (согласно правилу (1)), а V начинается на нетерминал K, то SJ принадлежит Лев(K).

(3) Если SJ входит в Лев(L) для некоторого L, причём L → V — правило грамматики и V начинается на нетерминал K, то SJ принадлежит Лев(K).

Заметим, что правило (3) можно рассматривать как аналог правила (2): в указанных в (3) предположениях ситуация L → _V согласована с SJ, а V начинается на нетерминал K.

Корректность этих правил в общем-то очевидна, если хорошенько подумать. Единственное, что требует некоторых пояснений — это то, почему с помощью правил (2) и (3) обнаружатся *все* терминалы K, для которых SJ принадлежит Лев(K). Попытаемся это объяснить. Рассмотрим правый вывод, в котором SJ стоит слева от K. Откуда мог взяться в нем нетерминал K? Если правило, которое его породило, породило также и конец слова SJ, то принадлежность SJ к Лев(K) будет обнаружена по правилу (2). Если же K было первой буквой слова, порождённого каким-то другим нетерминалом L, то — благодаря правилу (3) — достаточно установить принадлежность SJ к Лев(L). Осталось применить те же рассуждения к L и так далее.

В терминах LR-процесса то же самое можно сказать так. Сначала нетерминал K может участвовать в нескольких свёртках, не затрагивающих SJ (они соответствуют применению правила (3)), но затем он обязан подвергнуться свёртке, затрагивающей SJ (что соответствует применению правила (2)).

Осталось выяснить, какие ситуации согласованы с пустым словом, то есть для каких нетерминалов K пустое слово принадлежит Лев(K). Это определяется по следующим правилам:

(1) начальный нетерминал таков;

(2) если K таков и K → V — правило грамматики, причем слово V начинается с нетерминала L, то и L таков.

14.1.10. Проделать описанный анализ для грамматики

```
E  →  E + T
E  →  T
T  →  T * F
T  →  F
F  →  x
F  →  ( E )
```

(задающей тот же язык, что и грамматика примера 3, с. 223).

Решение. Множества Сост(S) для различных S приведены в таблице. Знак равенства означает, что множества ситуаций, являющиеся значениями функции Сост(S) на словах, стоящих слева и справа от знака равенства, одинаковы.

Правило определения Сост(SJ), если известны Сост(S) и J (здесь S — слово из терминалов и нетерминалов, J — терминал или нетерминал), таково:

> надо найти Сост(S) в правой колонке, взять соответствующее ему слово T в левой колонке, приписать к нему J и взять множество, стоящее напротив слова TJ (если слово TJ в таблице отсутствует, то Сост(SJ) пусто).

14.2. LR(0)-грамматики

Напомним, что наша основная цель — это поиск вывода заданного слова, или, другими словами, поиск успешного LR-процесса над ним. Во всех рассматриваемых нами грамматиках успешный LR-процесс (над данным словом) единствен. Искать этот единственный успешный процесс мы будем постепенно: в каждый момент мы смотрим, какой шаг возможен следующим. Для этого на грамматику надо наложить дополнительные требования, и сейчас мы рассмотрим простейший случай так называемых LR(0)-грамматик. Мы уже знаем:

> (1) В успешном LR-процессе возможна свёртка по правилу K → U при содержимом стека S тогда и только тогда, когда S при-

Слово S	Сост(S)
пустое	E → _E+T E → _T T → _T*F T → _F F → _x F → _(E)
E	E → E_+T
T	E → T_ T → T_*F
F	T → F_
x	F → x_
(	F → (_E) E → _E+T E → _T T → _T*F T → _F F → _x F → _(E)
E+	E → E+_T T → _T*F T → _F F → _x F → _(E)
T*	T → T*_F F → _x F → _(E)
(E	F → (E_) E → E_+T
(T	= T
(F	= F
(x	= x
((	= (
E+T	E → E+T_ T → T_*F
E+F	= F
E+x	= x
E+(	= (
T*F	T → T*F_
T*x	= x
T*(	= (
(E)	F → (E)_
(E+	= E+
E+T*	= T*

> надлежит ЛевКонт(K → U) или, другими словами, когда слово S согласовано с ситуацией K → U_.

Аналогичное утверждение про сдвиг гласит:

> (2) В успешном LR-процессе при содержимом стека S возможен сдвиг с очередным символом a тогда и только тогда, когда S согласовано с некоторой ситуацией K → U_aV.

14.2.1. Докажите это.

[Указание. Пусть произошел сдвиг и к стеку S добавилась буква a. Рассмотрите первую свёртку, затрагивающую эту букву.]

Рассмотрим некоторую грамматику и произвольное слово S из терминалов и нетерминалов. Если множество Сост(S) содержит ситуацию, в которой справа от подчёркивания стоит терминал, то говорят, что *для слова S возможен сдвиг*. Если в Сост(S) есть ситуация, в которой справа от подчёркивания ничего нет, то говорят, что *для слова S возможна свёртка* (по соответствующему правилу). Говорят, что для слова S возникает *конфликт типа сдвиг/свёртка*, если возможны и сдвиг, и свёртка. Говорят, что для слова S возникает *конфликт типа свёртка/свёртка*, если есть несколько правил, по которым возможна свёртка.

Грамматика называется LR(0)-*грамматикой*, если в ней нет конфликтов типа сдвиг/свёртка и свёртка/свёртка ни для одного слова S.

14.2.2. Является ли приведённая выше грамматика LR(0)-грамматикой?

Решение. Нет, не является. Для слов T и E+T имеются конфликты типа сдвиг/свёртка.

14.2.3. Являются ли LR(0)-грамматиками такие:

(а)
T	→	0
T	→	T1
T	→	TT2
T	→	TTT3

(б)
T	→	0
T	→	1T
T	→	2TT
T	→	3TTT

Решение. Являются, см. соответствующие таблицы (конфликтов нет).

Слово S	Сост(S)
пустое	T → _0 T → _T1 T → _TT2 T → _TTT3
0	T → 0_
T	T → T_1 T → T_T2 T → T_TT3
	T → _0 T → _T1 T → _TT2 T → _TTT3
T1	T → T1_
TT	T → TT_2 T → TT_T3
	T → T_1 T → T_T2 T → T_TT3
	T → _0 T → _T1 T → _TT2 T → _TTT3
TT2	T → TT2_
TTT	T → TTT_3 T → TT_2 T → TT_T3
	T → T_1 T → T_T2 T → T_TT3
	T → _0 T → _T1 T → _TT2 T → _TTT3
TT0	= 0
TTT3	T → TTT3_
TTT2	= TT2
TTTT	= TTT
TTT0	= 0

(а)

Слово S	Сост(S)
пустое	T → _0 T → _1T T → _2TT T → _3TTT
0	T → 0_
1	T → 1_T
	T → _0 T → _1T T → _2TT T → _3TTT
2	T → 2_TT
	T → _0 T → _1T T → _2TT T → _3TTT
3	T → 3_TTT
	T → _0 T → _1T T → _2TT T → _3TTT

(б), начало

Слово S	Сост(S)
1T	T → 1T_
10	= 0
11	= 1
12	= 2
13	= 3
2T	T → 2T_T
	T → _0 T → _1T T → _2TT T → _3TTT
20	= 0
21	= 1
22	= 2
23	= 3
3T	T → 3T_TT
	T → _0 T → _1T T → _2TT T → _3TTT
30	= 0
31	= 1
32	= 2
33	= 3
2TT	T → 2TT_
2T0	= 0
2T1	= 1
2T2	= 2
2T3	= 3
3TT	T → 3TT_T
	T → _0 T → _1T T → _2TT T → _3TTT
3T0	= 0
3T1	= 1
3T2	= 2
3T3	= 3
3TTT	T → 3TTT_
3TT0	= 0
3TT1	= 1
3TT2	= 2
3TT3	= 3

(б), окончание

Эта задача показывает, что LR(0)-грамматики могут быть как леворекурсивными, так и праворекурсивными.

14.2.4. Пусть дана LR(0)-грамматика. Доказать, что у любого слова существует не более одного правого вывода. Построить алгоритм проверки выводимости в LR(0)-грамматике.

Решение. Пусть дано произвольное слово. Будем строить LR-процесс над ним по шагам. Пусть текущее состояние стека LR-процесса равно `S`. Нам надо решить, делать сдвиг или свёртку (и если свёртку, то по какому правилу). Согласно определнию LR(0)-грамматики, в нашем состоянии `S` возможен либо только сдвиг, либо только свёртка (причем лишь по одному правилу). Таким образом, поиск возможных продолжений LR-процесса происходит детерминированно (на каждом шаге можно определить, какое действие только и возможно).

14.2.5. Что произойдет, если анализируемое слово не имеет вывода в данной грамматике?

Ответ. Либо на некотором шаге не будет возможен ни сдвиг, ни свёртка, либо все возможные сдвиги будет иметь неподходящий очередной символ.

Замечания. 1. При реализации этого алгоритма нет необходимости каждый раз заново вычислять множество Сост(`S`) для текущего значения `S`. Эти множества можно также хранить в стеке (в каждый момент хранятся множества Сост(`T`) для всех начал `T` текущего слова `S`).

2. На самом деле само слово `S` можно не хранить — достаточно хранить множества ситуаций Сост(`T`) для всех его начал `T` (включая само `S`).

В алгоритме проверки выводимости в LR(0)-грамматике мы используем не всю информацию, которую могли бы. В этом алгоритме для каждого состояния известно заранее, что в нем возможен только сдвиг или только свёртка (причем в последнем случае известно, по какому правилу). Более изощрённый алгоритм мог бы принимать решение о выборе между сдвигом и свёрткой, посмотрев на очередной символ (`Next`). Глядя на состояние, можно сказать, при каких значениях `Next` возможен сдвиг (это те терминалы, которые в ситуациях этого состояния стоят непосреджственно за подчёркиванием). Сложнее воспользоваться информацией о символе `Next` для решения вопроса о том, возможна ли свёртка. Для этого

есть упрощённый метод (грамматики, к которым он применим, называют SLR(1)-грамматиками [сокращение от Simple LR(1)]) и полный метод (более сложный, но использующий всю возможную информацию; грамматики, к которым он применим, называют LR(1)-грамматиками). Есть и промежуточный класс грамматик, называемый LALR(1).

14.3. SLR(1)-грамматики

Напомним, что для любого нетерминала K мы определяли (с. 242) множество Послед(K) тех терминалов, которые могут стоять непосредственно за K в выводимом (из начального нетерминала) слове; в это множество добавляют также символ EOI, если нетерминал K может стоять в конце выводимого слова.

14.3.1. Доказать, что если в данный момент LR-процесса последний символ стека S равен K, причем процесс этот может в дальнейшем успешно завершиться, то Next принадлежит Послед(K).

Решение. Этот факт является непосредственным следствием определения (вспомним соответствие между правыми выводами и LR-процессами).

Рассмотрим некоторую грамматику, произвольное слово S из терминалов и нетерминалов и терминал x. Если множество Сост(S) содержит ситуацию, в которой справа от подчёркивания стоит терминал x, то говорят, что *для пары* $\langle$S, x$\rangle$ *возможен сдвиг*. Если в Сост(S) есть ситуация K $\to$ U_, причем x принадлежит Послед(K), то говорят, что *для пары* $\langle$S, x$\rangle$ SLR(1)-*возможна свёртка* (по правилу K $\to$ U). Говорят, что для пары $\langle$S, x$\rangle$ возникает SLR(1)-*конфликт типа сдвиг/свёртка*, если возможны и сдвиг, и свёртка. Говорят, что для пары $\langle$S, x$\rangle$ возникает SLR(1)-*конфликт типа свёртка/свёртка*, если есть несколько правил, по которым возможна свёртка.

Грамматика называется SLR(1)-грамматикой, если в ней нет SLR(1)-конфликтов типа сдвиг/свёртка и свёртка/свёртка ни для одной пары $\langle$S, x$\rangle$.

14.3.2. Пусть дана SLR(1)-грамматика. Доказать, что у любого слова существует не более одного правого вывода. Построить алгоритм проверки выводимости в SLR(1)-грамматике.

Решение. Аналогично случаю LR(0)-грамматик, только при выборе между сдвигом и свёрткой учитывается очередной символ (`Next`).

14.3.3. Проверить, является ли приведенная выше на с. 254 грамматика (с нетерминалами `E`, `T` и `F`) SLR(1)-грамматикой.

Решение. Да, является, так как оба конфликта, мешающие ей быть LR(0)-грамматикой, разрешаются с учетом очередного символа: и для слова `T`, и для слова `E+T` сдвиг возможен только при $\texttt{Next} = \texttt{*}$, а символ `*` не принадлежит ни $\text{Послед}(\texttt{E}) = \{\texttt{EOI}, \texttt{+}, \texttt{)}\}$, ни $\text{Послед}(\texttt{T}) = \{\texttt{EOI}, \texttt{+}, \texttt{*}, \texttt{)}\}$, и поэтому при $\texttt{Next} = \texttt{*}$ свёртка невозможна.

14.4. LR(1)-грамматики, LALR(1)-грамматики

Описанный выше SLR(1)-подход используют не всю возможную информацию при выяснении того, возможна ли свёртка. Именно, он отдельно проверяет, возможна ли свёртка при данном состоянии стека `S` и отдельно — возможна ли свёртка по данному правилу при данном символе `Next`. Между тем эти проверки не являются независимыми: обе могут дать положительный ответ, но тем не менее свёртка при стеке `S` и очередном символе `Next` невозможна. В LR(1)-подходе этот недостаток устраняется.

LR(1)-подход состоит вот в чём: все наши определения и утверждения модифицируются так, чтобы учесть, какой символ стоит справа от разворачивамого нетерминала (другими словами, чему равен `Next` при свёртке).

Пусть $\texttt{K} \to \texttt{U}$ — одно из правил грамматики, а `t` — некоторый терминал или спецсимвол `EOI` (который мы домысливаем в конце входного слова). Определим множество $\text{ЛевКонт}(\texttt{K} \to \texttt{U}, \texttt{t})$ как множество всех слов, которые являются содержимым стека непосредственно перед свёрткой `U` в `K` в ходе успешного LR-процесса, при условии $\texttt{Next} = \texttt{t}$ (в момент свёртки).

Если отбросить у всех слов из $\text{ЛевКонт}(\texttt{K} \to \texttt{U})$ их конец `U`, то получится множество всех слов, которые могут появиться в правых выводах перед нетерминалом `K`, за которым стоит символ `t`. Это множество (не зависящее от того, какое из правил $\texttt{K} \to \texttt{U}$ для нетерминала `K` выбрано) мы будем обозначать $\text{Лев}(\texttt{K}, \texttt{t})$.

14.4.1. Написать грамматику для порождения множеств $\text{Лев}(\texttt{K}, \texttt{t})$.

Решение. Её нетерминалами будут символы $\langle$ЛевK t$\rangle$ для каждого нетерминала K и для каждого терминала t (а также для t = EOI). Её правила таковы. Пусть P — начальный нетерминал исхолной грамматики. Тогда в новой грамматике будет правило

$$\langle \text{ЛевP EOI} \rangle \to \qquad \text{(пустое слово)}.$$

Каждое правило исходной грамматики порождает несколько правил новой. Например, для правила

$$\text{K} \to \text{L u M N}$$

(L, M, N — нетерминалы, u — терминал) в новую грамматику мы добавим правила

$$\langle \text{ЛевL u} \rangle \to \langle \text{ЛевK x} \rangle$$

(для всех терминалов x);

$$\langle \text{ЛевM s} \rangle \to \langle \text{ЛевK y} \rangle \text{ L u}$$

(для всех s, которые могут начинать слова, выводимые из N, и для всех y, а также для всех пар s = y, если из N выводимо пустое слово);

$$\langle \text{ЛевN s} \rangle \to \langle \text{ЛевK s} \rangle \text{ L u M}$$

(для всех теминалов s).

14.4.2. Как меняется определение ситуации?

Решение. Ситуацией называется пара

[ситуация в старом смысле, терминал или EOI]

14.4.3. Как изменится определение согласованности?

Решение. Слово S из терминалов и нетерминалов согласовано с ситуацией [K $\to$ U_V, t] (здесь t — терминал или EOI), если S кончается на U, то есть S = TU, и, кроме того, T принадлежит Лев(K, t).

14.4.4. Каковы правила для индуктивного вычисления множества Сост(S) ситуаций, согласованных с данным словом S?

Ответ.

(1) Если слово S согласовано с ситуацией [K → U_V, t], причём слово V начинается на букву J, то есть V = JW, то слово SJ согласовано с ситуацией [K → UJ_W, t].

Это правило полностью определяет все ситуации с непустой левой половиной (то есть не начинающиеся с подчёркивания), согласованные с SJ. Осталось определить, для каких нетерминалов K и терминалов t слово SJ принадлежит Лев(K, t). Это делается по двум правилам:

(2) Если ситуация [L → U_V, t] согласована с SJ (согласно правилу (1)), а V начинается на нетерминал K, то SJ принадлежит Лев(K, s) для всех терминалов s, которые могут начинать слова, выводимые из слова V \ K (слово V без первой буквы K), а также для s = t, если из V \ K выводится пустое слово.

(3) Если SJ входит в Лев(L, t) для некоторых L и t, причем L → V — правило грамматики и V начинается на нетерминал K, то SJ принадлежит Лев(K, s) для всех терминалов s, которые могут начинать слова, выводимые из V \ K, а также для s = t, если из V \ K выводится пустое слово.

14.4.5. Дать определения конфликтов сдвиг/свёртка и свёртка/свёртка по аналогии с данными выше.

Решение. Пусть дана некоторая грамматика. Пусть S — произвольное слово из терминалов и нетерминалов. Если множество Сост(S) содержит ситуацию, в которой справа от подчёркивания стоит терминал t, то говорят, что для пары ⟨S, t⟩ возможен сдвиг. (Это определение не изменилось по сравнению с SLR(1)-случаем — вторые компоненты пар из Сост(S) не учитываются.)

Если в Сост(S) есть ситуация, в которой справа от подчёркивания ничего нет, а вторым членом пары является терминал t, то говорят, что для пары ⟨S, t⟩ LR(1)-*возможна свёртка* (по соответствующему правилу). Говорят, что для пары ⟨S, t⟩ возникает LR(1)-*конфликт типа сдвиг/свёртка*, если возможны и сдвиг, и свёртка. Говорят, что для пары ⟨S, t⟩ возникает LR(1)-*конфликт типа свёртка/свёртка*, если есть несколько правил, по которым возможна свёртка.

Грамматика называется LR(1)-*грамматикой*, если в ней нет LR(1)-конфликтов типа сдвиг/свёртка и свёртка/свёртка ни для одной пары $\langle \mathtt{S}, \mathtt{t} \rangle$

14.4.6. Построить алгоритм проверки выводимости слова в LR(1)-грамматике.

Решение. Как и раньше, на каждом шаге LR-процесса можно однозначно определить, какой шаг только и может быть следующим.

Полезно (в частности, для LALR(1)-разбора, смотри ниже) понять, как связаны понятия LR(0) и LR(1)-согласованности.

14.4.7. Сформулировать и доказать соответствующее утверждение.

Ответ. Пусть фиксирована некоторая грамматика. Слово S из терминалов и нетерминалов является LR(0)-согласованным с ситуацией $\mathtt{K} \to \mathtt{U_V}$ тогда и только тогда, когда оно LR(1)-согласовано с парой $[\mathtt{K} \to \mathtt{U_V}, \mathtt{t}]$ для некоторого терминала t (или для $\mathtt{t} = \mathtt{EOI}$). То же самое другими словами: Лев(K) есть объединение Лев(K, t) по всем t. В последней форме это совсем ясно.

Замечание. Таким образом, функция Сост(S) в LR(1)-смысле является расширением функции Сост(S) в LR(0)-смысле: $\text{Сост}_{\text{LR(0)}}(\mathtt{S})$ получается из $\text{Сост}_{\text{LR(1)}}(\mathtt{S})$, если во всех парах выбросить вторые члены.

Теперь мы можем дать определение LALR(1)-грамматики. Пусть фиксирована некоторая грамматика, S — слово из нетерминалов и терминалов, t — некоторый терминал (или EOI). Будем говорить, что для пары $\langle \mathtt{S}, \mathtt{t} \rangle$ LALR(1)-возможна свёртка по некоторому правилу, если существует другое слово $\mathtt{S}_1$ с $\text{Сост}_{\text{LR(0)}}(\mathtt{S}_0) = \text{Сост}_{\text{LR(0)}}(\mathtt{S}_1)$, причем для пары $\langle \mathtt{S}_1, \mathtt{t} \rangle$ LR(1)-возможна свёртка по рассматриваемому правилу. Далее определяются конфликты (естественным образом), и грамматика называется LALR(1)-грамматикой, если конфликтов нет.

14.4.8. Доказать, что всякая SLR(1)-грамматика является LALR(1)-грамматикой, а всякая LALR(1)-грамматика является LR(1)-грамматикой.

Указание. Это — простое следствие определений.

14.4.9. Построить алгоритм проверки выводимости в LALR(1)-грамматике, который хранит в стеке меньше информации, чем соответствующий LR(1)-алгоритм.

Указание. Достаточно хранить в стеке множества $\text{Сост}_{\text{LR}(0)}(\mathtt{S})$, поскольку согласно определению LALR(1)-возможность свёртки ими определяется. (Так что сам алгоритм ничем не отличается от SLR(1)-случая, кроме таблицы возможных свёрток.)

14.4.10. Привести пример LALR(1)-грамматики, не являющейся SLR(1)-грамматикой.

14.4.11. Привести пример LR(1)-грамматики, не являющейся LALR(1)-грамматикой.

14.5. Общие замечания о разных методах разбора

Применение этих методов на практике имеет свои хитрости и тонкости, которых мы не касались. (Например, таблицы следует хранить по возможности экономно.) Часто оказывается также, что для некоторого входного языка наиболее естественная грамматика не является LL(1)-грамматикой, но является LR(1)-грамматикой, а также может быть заменена на LL(1)-грамматику без изменения языка. Какой из этих вариантов выбрать, не всегда ясно. Дилетантский совет: если Вы сами проектируете входной язык, то не следует выпендриваться и употреблять одни и те же символы для разных целей — и тогда обычно несложно написать LL(1)-грамматику или рекурсивный анализатор. Если же входной язык задан заранее с помощью LR(1)-грамматики, не являющейся LL(1)-грамматикой, то лучше её не трогать, а разбирать как есть. При этом могут оказаться полезные средства автоматического порождения анализаторов, наиболее известными из которых являются `yacc` (UNIX) и `bison` (GNU).

Большое количество полезной и хорошо изложенной информации о теории и практике синтаксического разбора имеется в книге: Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. Compilers: principles, techniques and tools. Addison-Wesley (1985).

Предметный указатель

backtracking, 62
Borland, 2

LALR(1)-грамматика, 264
LL(1)-грамматика, 242
LL(1)-разбор, 239
LR(0)-грамматика, 256
LR(1)-грамматика, 264
LR-процесс, 248

NP-полнота, 74

SLR(1)-грамматика, 260

Turbo Pascal, 2, 29

АВЛ-дерево, 207
автомат
    конечный, 90, 165, 178
    конечный недетерминированный, 184
Адельсон-Вельский, Г.М., 207
алфавит, 167
Ахо (Aho, A.V.), 74, 187, 265

Б-дерево, 218
билеты счастливые, число, 60
биномиальный коэффициент, 141
ближайшая сумма, 32
Бойера–Мура алгоритм, 174
Брудно, А.Л., 25
быстрое умножение, 28

Вайнцвайг, М.Н., 41
Варсанофьев, Д.В., 40, 196
вершина графа, 109, 136
ветвей и границ метод, 62
вращение
    левое, правое, 208
    малое, большое, 208
вывод
    в грамматике, 220
    левый, 240
    правый, 248
выводимость в КС-грамматике
    полиномиальный алгоритм, 224
выпуклая оболочка, 85, 113
выражение, 223
    регулярное, 182
высота, 199

гауссовы числа, 16
голландский флаг, 38
Горнера схема, 26
грамматика
    LALR(1), 264
    LL(1), 242
    LR(1), 264
    LR(0), 256
    SLR(1), 260
    выражений, 223
    контекстно-свободная, 220
    леворекурсивная, 243
граф
    вершина, 109, 136

двудольный, 162
кратчайшие пути, 154
неориентированный, 136
ориентированный, 109
ребро, 136
связная компонента, 117, 136, 158
связный, 109
Грея коды, 50
Грис (Gries, D.), 25, 32, 35, 41

датчик поворота, 52
двоичный поиск, 34
двудольный граф, 162
Дейкстра (Dijkstra, E.W.), 11, 21, 38
Дейкстры алгоритм (кратчайшие пути), 156, 158
дек
реализация в массиве, 107
ссылочная реализация, 113
деление с остатком, 8
быстрое, 22
Део (Deo, N.), 219
дерево
Б-дерево, 218
вершина, 125
высота, 127
двоичное, 79
корень, 125
обход, 63, 127, 132, 149
обход нерекурсивный, 148
позиций, 62
позиций, реализация, 70
полное двоичное, 198
рекурсивная обработка, 126
сбалансированное, 207
ссылочная реализация, 125, 200
упорядоченное, 199
число вершин, 126, 149
число листьев, 127
десятичная дробь, период, 20
десятичная запись
печать, 16, 19, 148
печать рекурсивная, 123
чтение, 92
детерминизация конечного автомата, 185
Дементман, А.М., 41
динамическое программирование, 141, 143
кратчайшие пути, 154
диофантово уравнение, 11, 13
дополнение
регулярного множества, 187

Евклида алгоритм, 10, 11
двоичный, 12

жулик на пособии, 32

задача
NP-полная, 74
о рюкзаке, 73, 146
Звонкин, А.К., 148
Звонкин, Д., 13

индуктивная функция, 38
индуктивное расширение, 39
источник, 184

калькулятор

стековый, 152
Каталана числа, 56, 143
Кнута–Морриса–Пратта алгоритм, 170
коды Грея, 50
количество различных, 24, 25, 84
комментарии
вложенные, 91
удаление, 90
конец, 167
конечный автомат, 90, 165, 178
недетерминированный, 184
контекстно-свободная грамматика, 220
конфликт
свёртка/свёртка, 256, 260, 263
сдвиг/свёртка, 256, 260, 263
коэффициент
биномиальный, 141
КС-грамматика, 220
Кушниренко, А.Г., 25, 27, 38, 107

Ландис, Е.М., 207
Лев(K), 250
Лев(K, t), 261
ЛевКонт(K → U), 249
ЛевКонт(K → U, t), 261
левый контекст правила, 249
Лисовский, Анджей, 148

массив, 22
с минимальным элементом, 119
матриц порядок умножения, 145
матриц произведение, 157
матрица цен, 157
медиана, поиск, 88, 139
минимум, поиск, 87
многоугольника триангуляция, 59
многочлен
значение, 26
производная, 27
умножение, 27, 28
множество
представление, 188, 192
представление деревом, 198
реализация в битовом массиве, 115
реализация перечнем, 116
регулярное, 183
тип данных, 115
множитель, 223
моделирование, очередь событий, 120
монотоных последовательностей перечисление, 47

наибольший общий делитель, 9
наименьшее общее кратное, 11
Напр(K → V), 242
Нач(X), 228, 241
начало, 167
неассоциативное произведение, 146
недетерминированный конечный автомат, 184
неориентированный граф, 136
нетерминал, 220

Нивергельт (Nievergelt, J.), 219
НОД, 9
НОК, 11

обмен значений, 6
образец, поиск, 165
обратная перестановка, 36
обратная польская запись, 151
обход дерева, 62
    рекурсивный, 132
общий элемент (в упорядоченных массивах), 32
опечатки, поиск, 196
ориентированный граф, 109
орфография, проверка, 196
открытая адресация, 188
очередь, 105
    из двух стеков, 107
    приоритетная, 119
    реализация в массиве, 105
    ссылочная реализация, 111
ошибка
    индекс за границей, 29, 35, 36, 76, 79

паскаль, 1, 29
Паскаль (Pascal, B.), 1, 142
Паскаля треугольник, 59, 142
перебор с возвратами, 62
пересечение
    регулярных множеств, 187
    упорядоченных массивов, 31
перестановка
    обратная, 36, 54
    чётность, 36
перестановка частей массива, 25
перестановок
    перечисление, 44, 53, 129
период десятичной дроби, 20
поддерево, 199
подмножеств
    данного размера перечисление, 45
    перечисление, 44
подпоследовательность
    максимальная возрастающая, 41
    общая, 41
    проверка, 40
подслово, 167
    поиск, 170, 173, 174, 177
поиск
    k-го по порядку, 88, 139, 206
    в глубину, 160
    в ширину, 118, 158
    двоичный, 34
    кратчайшего пути, 154
    минимума, 87
    образца, 174, 177, 178
    одного из слов, 180
    подслова, 165, 170, 173, 174, 177
    представителя большинства, 88
Посл(X), 228
Послед(X), 242
последовательности монотонные
    перечисление, 130
последовательность
    содержащая все слова длины

n, 110
постфиксная запись, 151
потомок, вершины, 125
права, авторские, 2
приведение, 248
приоритетная очередь, 119
программирование
  динамическое, 141, 143, 224
произведение
  многочленов, 27
  неассоциативное, 58, 146
простые множители, 14
путей число, 157
Пятьдесят седьмая школа, 1

Рабина алгоритм, 177
разбиений на слагаемые
  перечисление, 48, 131
  число, 59
разбор
  LL(1), 239
  LR(1), 248
  общий КС-алгоритм, 224
  рекурсивный спуск, 226
разложение на множители, 14
размещения с повторениями, 43, 128
расстановки функция, 188
расширение индуктивное, 39
ребро графа, 136
регулярное
  выражение, 182
  множество, 183
регулярное множество
  дополнение, 187
  пересечение, 187
Рейнгольд (Reingold, E.M.), 219
рекурсивная программа, 121
рекурсивный спуск, 226
рекурсия, 121
  устранение, 141
рюкзак, заполнение, 73, 146

свёртка, 248
связная компонента
  неориентированного графа, 136
  ориентированного графа, 117, 138, 158
связный граф, 109
сдвиг, 248
Сети (Sethi, R.), 187, 265
символ
  начальный, 220
  нетерминальный, 220
  терминальный, 220
ситуация
  грамматики, 251
скобки
  правильность, 99, 220
скобок расстановка, 58
слагаемое, 223
слияние (упорядоченных массивов), 30
слово, 167
  выводимое, 220
сортировка
  $n \log n$, 76
  деревом, 79, 120
  квадратичная, 75
  нижняя оценка сложности, 85

слиянием, 76, 83
топологическая, 133, 162
Хоара (быстрая), 83, 138
Хоара, нерекурсивная, 150
цифровая, 86
число сравнений, 85
Сост(S), 252
составные символы, замена, 90
сочетаний
число, 59, 141
стек, 97
два в массиве, 101
отложенных заданий, 147
реализация в массиве, 97
ссылочная реализация, 102
стековый калькулятор, 152
степень
быстрое вычисление, 7
вычисление, 6
рекурсивная программа, 122
счастливые билеты, число, 60

терминал, 220
направляющий, 242
топологическая сортировка, 133, 162
треугольник Паскаля, 142
триангуляция многоугольника, 59, 143

Ульман (Ullman, J.D.), 74, 187, 265
упорядоченное дерево, 199

факториал, 9
рекурсивная программа, 121
ферзей расстановка, 62
Фибоначчи последовательность, 9, 143, 207
быстрое вычисление, 9
Флойда алгоритм, 155, 186
Форда–Беллмана алгоритм, 154
функция
индуктивная, 38
расстановки, 188

ханойские башни
нерекурсивная программа, 147
рекурсивная программа, 124
хеш-функция, 188
универсальное семейство, 194, 195
хеширование, 188
оценка числа действий, 194
с открытой адресацией, 188
со списками, 192
универсальное, 194
Хоар (Hoare, C.A.R.), 1, 138, 150
Хоара сортировка, 83, 138
нерекурсивная, 150
Хопкрофт (Hopcroft, J.), 74
хорды окружности, 59

целые точки в круге, 17
цикл, эйлеров (по всем рёбрам), 109

чётность перестановки, 36
число
общих элементов, 28

разбиений, 59
сочетаний, 59
счастливых билетов, 60

язык
контекстно-свободный, 220
не контекстно-свободный, 222