585. Динамические линейные уравнения. [Dynamic linear equations] Пользователи METAFONT'а задают значения переменных неявно, записывая уравнения, которым они должны удовлетворять; ЭВМ должна быть достаточно смышлёной, чтобы решать эти уравнения. И в самом деле, ЭВМ отважно пытается это сделать, выделив пять различных типов числовых значений:

type(p) = *known* это приятный случай, когда *value*(p) — масштабное [*scaled*] значение переменной, адрес которой суть p.

type(p) = *dependent* означает, что *value*(p) отсутствует, но *dep_list*(p) указывает на *список зависимостей*, выражающий значение переменной p, как число типа *scaled* плюс сумма независимых переменных с коэффициентами типа *fraction*.

type(p) = *independent* означает, что *value*$(p) = 64s + m$, где $s > 0$ — «порядковый номер» ["serial number"], отражающий время, когда эта переменная первый раз использовалась в уравнении; кроме того $0 \leqslant m < 64$. Каждая зависимая переменная, ссылающаяся на этот номер, в действительности ссылается на будущее значение этой переменной, умноженное на 2^m. (Обычно $m = 0$, но более высокие степени масштабирования иногда нужны, чтобы уберечь коэффициенты в списке зависимостей от получения слишком больших значений. Значение m всегда будет чётным.)

type(p) = *numeric_type* означает, что переменная p не появлялась ранее в уравнении, но она была явно объявлена как числовая.

type(p) = *undefined* означает, что переменная p ни когда ранее не появлялась.

Мы обсудили эти пять типов в обратном порядке истории [их преобразований] во время вычисления: переменная, однажды ставшая известной *known*, никогда не станет опять зависимой *dependent*; переменная, однажды ставшая зависимой *dependent*, почти никогда не станет снова независимой *independent*; переменная, ставшая независимой *independent*, никогда не станет опять числового типа *numeric_type*; а однажды ставшая числового типа *numeric_type*, никогда не станет снова неопределённой *undefined* (кроме, конечно, случаев, когда пользователь сам решил отбросить старое значение и начать [расчёт] снова). Тем не менее, обратный переход может иметь место: иногда зависимая переменная *dependent* опять становится независимой *independent*, когда одна из независимых переменных, от которых она зависит, возвращается к типу неопределённой *undefined*.

```
define s_scale = 64  { порядковые номера умножаются на этот коэффициент }
define new_indep(#) ≡  { создать новую независимую переменную }
    begin type(#) ← independent; serial_no ← serial_no + s_scale; value(#) ← serial_no;
    end
```

⟨ Общие переменные 13 ⟩ +≡
```
serial_no: integer;  { последний порядковый номер, умноженный на s_scale }
```

586. ⟨ Сделай переменную $q + s$ вновь независимой 586 ⟩ ≡
```
new_indep(q + s)
```
Этот код используется в разделе 232.

587. Но как представляются списки зависимостей? Это просто: линейная комбинация $\alpha_1 v_1 + \cdots + \alpha_k v_k + \beta$ появляется в $k+1$ узлах значений. Если $q = \mathit{dep\_list}(p)$ указывает на этот список, и если $k > 0$, то $\mathit{value}(q) = \alpha_1$ (которое имеет тип *fraction*); $\mathit{info}(q)$ указывает на место [величины] v_1; и $\mathit{link}(p)$ указывает на список зависимостей $\alpha_2 v_2 + \cdots + \alpha_k v_k + \beta$. С другой стороны, если $k = 0$, то $\mathit{value}(q) = \beta$ (которое является типа *scaled*) и $\mathit{info}(q) = \mathit{null}$. Независимые переменные v_1, ..., v_k сохранены так, чтобы они появлялись в убывающем порядке их полей *value* (то есть, их порядковых номеров). (Удобно использовать убывающий порядок, т. к. $\mathit{value}(\mathit{null}) = 0$. Если бы независимые переменные сохранялись бы не по их порядковым номерам, а по каким-либо иным признакам, например, по их положению в массиве *mem*, механизм решения уравнений был бы слишком системно-зависимым, потому что упорядочивание может влиять на вычисленные значения.)

Поле *link* в узле, содержащем постоянный член β, называется *последней ссылкой [final link]* списка зависимостей. METAFONT поддерживает двунаправленный главный список [master list] всех списков зависимостей с помощью постоянно размещённого в массиве *mem* узла *dep_head*. Если нет зависимостей, мы имеем $\mathit{link}(\mathit{dep\_head}) = \mathit{dep\_head}$ и $\mathit{prev\_dep}(\mathit{dep\_head}) = \mathit{dep\_head}$; иначе $\mathit{link}(\mathit{dep\_head})$ указывает на первую зависимую переменную, скажем p, и $\mathit{prev\_dep}(p) = \mathit{dep\_head}$. Мы имеем $\mathit{type}(p) = \mathit{dependent}$, а $\mathit{dep\_list}(p)$ указывает на её список зависимостей. Если самая последняя ссылка этого списка зависимостей встречается в месте q, то $\mathit{link}(q)$ указывает на следующую зависимую переменную (скажем r); и мы имеем $\mathit{prev\_dep}(r) = q$, и т. д.

define $\mathit{dep\_list}(\#) \equiv \mathit{link}(\mathit{value\_loc}(\#))$ { половина поля *value* в переменной *dependent* }
define $\mathit{prev\_dep}(\#) \equiv \mathit{info}(\mathit{value\_loc}(\#))$ { другая половина; делает двусвязный список }
define $\mathit{dep\_node\_size} = 2$ { число слов в узле зависимостей [dependency node] }

⟨ Настрой таблицы (делает только INIMF) 176 ⟩ +≡
$\mathit{serial\_no} \leftarrow 0$; $\mathit{link}(\mathit{dep\_head}) \leftarrow \mathit{dep\_head}$; $\mathit{prev\_dep}(\mathit{dep\_head}) \leftarrow \mathit{dep\_head}$; $\mathit{info}(\mathit{dep\_head}) \leftarrow \mathit{null}$;
$\mathit{dep\_list}(\mathit{dep\_head}) \leftarrow \mathit{null}$;

588. На самом деле, приведённое выше описание содержит невинную ложь. Есть другой вид переменной, называемый *proto_dependent*, который очень похож на *dependent* с той разницей, что коэффициенты α в его списках зависимостей типа *scaled*, а не дроби. Списки протозависимостей перемешаны со списками зависимостей в узлах, достижимых из *dep_head*.

589. Вот процедура, печатающая список зависимостей в символьном виде. Второй параметр должен быть или *dependent*, или *proto_dependent*, чтобы указать масштабирование коэффициентов.

```
⟨ Объяви подпрограммы для печати выражений 257 ⟩ +≡
procedure print_dependency(p : pointer; t : small_number);
  label exit;
  var v: integer;  { коэффициент }
    pp, q: pointer;  { для работы со списком }
  begin pp ← p;
  loop begin v ← abs(value(p)); q ← info(p);
    if q = null then  { постоянный член }
      begin if (v ≠ 0) ∨ (p = pp) then
        begin if value(p) > 0 then
          if p ≠ pp then print_char("+");
        print_scaled(value(p));
        end;
      return;
      end;
    ⟨ Печатай коэффициент, если он не равен ±1.0 590 ⟩;
    if type(q) ≠ independent then confusion("dep");
    print_variable_name(q); v ← value(q) mod s_scale;
    while v > 0 do
      begin print("*4"); v ← v − 2;
      end;
    p ← link(p);
    end;
exit: end;
```

590.

```
⟨ Печатай коэффициент, если он не равен ±1.0 590 ⟩ ≡
  if value(p) < 0 then print_char("-")
  else if p ≠ pp then print_char("+");
  if t = dependent then v ← round_fraction(v);
  if v ≠ unity then print_scaled(v)
```

Этот код используется в разделе 589.

591. Наибольшее абсолютное значение коэффициента в данном списке зависимостей возвращается следующей простой функцией.

```
function max_coef(p : pointer): fraction;
  var x: fraction;  { максимум на данный момент }
  begin x ← 0;
  while info(p) ≠ null do
    begin if abs(value(p)) > x then x ← abs(value(p));
    p ← link(p);
    end;
  max_coef ← x;
  end;
```

592. Одно из основных действий, необходимых для списков зависимостей, — добавление элементов [multiple] одного списка к [элементам] другого; мы называем это *p_plus_fq*, где p и q указывают на списки зависимостей, а f — дробь [fraction].

Если коэффициент любой независимой переменной становится по модулю равным или больше значения *coef_bound*, то эта процедура заменяет её тип на '*independent_needing_fix*' и устанавливает глобальную переменную *fix_needed* в значение *true*. Значение *coef_bound* $= \mu$ выбирается так, чтобы $\mu^2 + \mu < 8$; это означает, что числа, с которыми мы имеем дело, не будут слишком больши́ми. (Вместо «оптимального» $\mu = (\sqrt{33} - 1)/2 \approx 2,3723$, в качестве порога принято более безопасное значение $7/3$.)

Упомянутые в предыдущем абзаце изменения на самом деле выполняются только, когда глобальная переменная *watch_coefs* установлена в значение *true*. Обычно, так оно и есть; по сути, она имеет значение *false* только, когда METAFONT делает список зависимостей, который вскоре будет приравнен нулю.

Несколько процедур, действующих на списки зависимостей, включая *p_plus_fq*, устанавливают глобальную переменную *dep_final* на последний узел (постоянный член) создаваемого ими списка зависимостей.

define *coef_bound* $\equiv$ *'4525252525* { *fraction* приближение к $7/3$ }
define *independent_needing_fix* $= 0$

⟨ Общие переменные 13 ⟩ +≡
fix_needed: *boolean*; { хотя бы одну назависимую [*independent*] переменную нужно масштабировать? }
watch_coefs: *boolean*; { нам нужно масштабировать коэффициенты, которые превышают *coef_bound*? }
dep_final: *pointer*; { место постоянного члена и последней ссылки }

593. ⟨ Установи начальные значения ключевых переменных 21 ⟩ +≡
fix_needed ← *false*; *watch_coefs* ← *true*;

594. Процедура *p_plus_fq* имеет четвёртый параметр t, который должен устанавливаться [в значение] *proto_dependent*, если p — список протозависимостей. В этом случае f будет типа *scaled*, а не типа *fraction*. Таким же образом, пятый параметр tt должен быть *proto_dependent*, если q — список протозависимостей.

Список q не изменяется этой операцией; но список p полностью разрушается.

Последняя ссылка списка зависимостей или списка протозависимостей, возвращаемая [функцией] *p_plus_fq*, та же, что и исходная последняя ссылка [списка] p. В самом деле, постоянный член итогового списка будет расположен в том же месте [массива] *mem*, что и исходный постоянный член [списка] p.

Коэффициенты итогового списка полагаются равными нулю, если они меньше определённого порогового значения. Это возмещает неизбежные ошибки округления, и стремится сделать больше переменных [известными] '*known*'. Порог приблизительно 10^{-5} для обычного списка зависимостей, 10^{-4} для протозависимостей.

```
define fraction_threshold = 2685  { коэффициент fraction, меньший этого, обнуляется }
define half_fraction_threshold = 1342  { половина fraction_threshold }
define scaled_threshold = 8  { коэффициент scaled, меньший этого, обнуляется }
define half_scaled_threshold = 4  { половина scaled_threshold }
```

⟨ Объяви основные подпрограммы для списков зависимостей 594 ⟩ ≡

```
function p_plus_fq(p : pointer; f : integer; q : pointer; t, tt : small_number): pointer;
  label done;
  var pp, qq: pointer;  { info(p) и info(q), соответственно }
    r, s: pointer;  { для работы со списком }
    threshold: integer;  { определяет окресность нуля }
    v: integer;  { временный регистр }
  begin if t = dependent then threshold ← fraction_threshold
  else threshold ← scaled_threshold;
  r ← temp_head; pp ← info(p); qq ← info(q);
  loop if pp = qq then
      if pp = null then goto done
      else ⟨ Внеси член из p, добавь f раз соответствующий член из q 595 ⟩
    else if value(pp) < value(qq) then ⟨ Внеси член из q, умноженный на f 596 ⟩
      else begin link(r) ← p; r ← p; p ← link(p); pp ← info(p);
        end;
done: if t = dependent then value(p) ← slow_add(value(p), take_fraction(value(q), f))
  else value(p) ← slow_add(value(p), take_scaled(value(q), f));
  link(r) ← p; dep_final ← p; p_plus_fq ← link(temp_head);
  end;
```

См. также разделы 600, 602, 603 и 604.

Этот код используется в разделе 246.

595. ⟨ Внеси член из p, добавь f раз соответствующий член из q 595 ⟩ ≡

```
begin if tt = dependent then v ← value(p) + take_fraction(f, value(q))
else v ← value(p) + take_scaled(f, value(q));
value(p) ← v; s ← p; p ← link(p);
if abs(v) < threshold then free_node(s, dep_node_size)
else begin if abs(v) ⩾ coef_bound then
    if watch_coefs then
      begin type(qq) ← independent_needing_fix; fix_needed ← true;
      end;
  link(r) ← s; r ← s;
  end;
pp ← info(p); q ← link(q); qq ← info(q);
end
```

Этот код используется в разделе 594.

596. ⟨ Внеси член из q, умноженный на f 596 ⟩ ≡

```
begin if tt = dependent then v ← take_fraction(f, value(q))
else v ← take_scaled(f, value(q));
if abs(v) > half(threshold) then
  begin s ← get_node(dep_node_size); info(s) ← qq; value(s) ← v;
  if abs(v) ⩾ coef_bound then
    if watch_coefs then
      begin type(qq) ← independent_needing_fix; fix_needed ← true;
      end;
  link(r) ← s; r ← s;
  end;
q ← link(q); qq ← info(q);
end
```

Этот код используется в разделе 594.

597. Удобно иметь другую подпрограмму для особого случая *p_plus_fq*, когда $f = 1.0$. В этой подпрограмме оба списка p и q одного и того же типа t (или *dependent*, или *proto_dependent*).

```
function p_plus_q(p : pointer; q : pointer; t : small_number): pointer;
  label done;
  var pp, qq: pointer;   { info(p) и info(q), соответственно }
    r, s: pointer;   { для работы со списком }
    threshold: integer;   { определяет окрестность нуля }
    v: integer;   { временный регистр }
  begin if t = dependent then threshold ← fraction_threshold
  else threshold ← scaled_threshold;
  r ← temp_head; pp ← info(p); qq ← info(q);
  loop if pp = qq then
      if pp = null then goto done
      else ⟨ Внеси член из p, добавь соответствующий член из q 598 ⟩
    else if value(pp) < value(qq) then
      begin s ← get_node(dep_node_size); info(s) ← qq; value(s) ← value(q); q ← link(q);
      qq ← info(q); link(r) ← s; r ← s;
      end
    else begin link(r) ← p; r ← p; p ← link(p); pp ← info(p);
      end;
done: value(p) ← slow_add(value(p), value(q)); link(r) ← p; dep_final ← p; p_plus_q ← link(temp_head);
  end;
```

598. ⟨ Внеси член из p, добавь соответствующий член из q 598 ⟩ ≡

```
begin v ← value(p) + value(q); value(p) ← v; s ← p; p ← link(p); pp ← info(p);
if abs(v) < threshold then free_node(s, dep_node_size)
else begin if abs(v) ⩾ coef_bound then
    if watch_coefs then
      begin type(qq) ← independent_needing_fix; fix_needed ← true;
      end;
  link(r) ← s; r ← s;
  end;
q ← link(q); qq ← info(q);
end
```

Этот код используется в разделе 597.

599. Немного более простая подпрограмма умножит список зависимостей на данную константу v. Эта константа или дробь [*fraction*], не превышающая *fraction_one*, или масштабное целое [*scaled*]. В последнем случае мы можем принудительно преобразовать список зависимостей в список протозависимостей. Параметры *t0* и *t1* суть типы списков до и после; они должны совпадать, если не [выполняются одновременно] $t0 = dependent$, $t1 = proto\_dependent$ и $v\_is\_scaled = true$.

```
function p_times_v(p : pointer; v : integer; t0, t1 : small_number; v_is_scaled : boolean): pointer;
  var r, s: pointer;   { для работы со списком }
    w: integer;   { предполагаемый коэффициент }
    threshold: integer; scaling_down: boolean;
  begin if t0 ≠ t1 then scaling_down ← true else scaling_down ← ¬v_is_scaled;
  if t1 = dependent then threshold ← half_fraction_threshold
  else threshold ← half_scaled_threshold;
  r ← temp_head;
  while info(p) ≠ null do
    begin if scaling_down then w ← take_fraction(v, value(p))
    else w ← take_scaled(v, value(p));
    if abs(w) ⩽ threshold then
      begin s ← link(p); free_node(p, dep_node_size); p ← s;
      end
    else begin if abs(w) ⩾ coef_bound then
        begin fix_needed ← true; type(info(p)) ← independent_needing_fix;
        end;
      link(r) ← p; r ← p; value(p) ← w; p ← link(p);
      end;
    end;
  link(r) ← p;
  if v_is_scaled then value(p) ← take_scaled(value(p), v)
  else value(p) ← take_fraction(value(p), v);
  p_times_v ← link(temp_head);
  end;
```

600. Таким же образом, нам иногда нужно разделить список зависимостей на данную константу типа *scaled*.

```
⟨ Объяви основные подпрограммы для списков зависимостей 594 ⟩ +≡
function p_over_v(p : pointer; v : scaled; t0, t1 : small_number): pointer;
  var r, s: pointer; { для работы со списком }
    w: integer; { предварительный коэффициент }
    threshold: integer; scaling_down: boolean;
  begin if t0 ≠ t1 then scaling_down ← true else scaling_down ← false;
  if t1 = dependent then threshold ← half_fraction_threshold
  else threshold ← half_scaled_threshold;
  r ← temp_head;
  while info(p) ≠ null do
    begin if scaling_down then
      if abs(v) < '2000000 then w ← make_scaled(value(p), v * '10000)
      else w ← make_scaled(round_fraction(value(p)), v)
    else w ← make_scaled(value(p), v);
    if abs(w) ⩽ threshold then
      begin s ← link(p); free_node(p, dep_node_size); p ← s;
      end
    else begin if abs(w) ⩾ coef_bound then
        begin fix_needed ← true; type(info(p)) ← independent_needing_fix;
        end;
      link(r) ← p; r ← p; value(p) ← w; p ← link(p);
      end;
    end;
  link(r) ← p; value(p) ← make_scaled(value(p), v); p_over_v ← link(temp_head);
  end;
```

601. Вот другая полезная подпрограмма для списков зависимостей. Когда независимая переменная становится зависимой, мы хотим удалить её из всех существующих зависимостей. Функция *p_with_x_becoming_q* вычисляет список зависимостей p после того, как переменная x замещается переменной q.

У этой процедуры, по сути, такие же соглашения о вызове, что и у *p_plus_fq*: список q не изменяется; список p уничтожается; узел постоянной [constant node] и последняя ссылка наследуются из p; а четвёртый параметр указывает, имеет ли список p тип *proto_dependent*. Тем не менее, глобальная переменная *dep_final* не изменяется, если x в списке p отсутствует.

```
function p_with_x_becoming_q(p, x, q : pointer; t : small_number): pointer;
  var r, s: pointer; { для работы со списком }
    v: integer; { коэффициент [величины] x }
    sx: integer; { порядковый номер [величины] x }
  begin s ← p; r ← temp_head; sx ← value(x);
  while value(info(s)) > sx do
    begin r ← s; s ← link(s);
    end;
  if info(s) ≠ x then p_with_x_becoming_q ← p
  else begin link(temp_head) ← p; link(r) ← link(s); v ← value(s); free_node(s, dep_node_size);
    p_with_x_becoming_q ← p_plus_fq(link(temp_head), v, q, t, dependent);
    end;
  end;
```

602. Вот простая процедура, сообщающая об ошибке, когда переменная получает известное значение [known value], выходящее за требуемые границы.

```
⟨ Объяви основные подпрограммы для списков зависимостей 594 ⟩ +≡
procedure val_too_big(x : scaled);
  begin if internal[warning_check] > 0 then
    begin print_err("Value␣is␣too␣large␣("); print_scaled(x); print_char(")");
    help4 ("The␣equation␣I␣just␣processed␣has␣given␣some␣variable")
    ("a␣value␣of␣4096␣or␣more.␣Continue␣and␣I'll␣try␣to␣cope")
    ("with␣that␣big␣value;␣but␣it␣might␣be␣dangerous.")
    ("(Set␣warningcheck:=0␣to␣suppress␣this␣message.)"); error;
    end;
  end;
```

603. Когда зависимая переменная становится известной, следующая подпрограмма удаляет её список зависимостей. Здесь p указывает на переменную, а q указывает на список зависимостей (длиной в один узел).

```
⟨ Объяви основные подпрограммы для списков зависимостей 594 ⟩ +≡
procedure make_known(p, q : pointer);
  var t: dependent .. proto_dependent; { Предыдущий тип }
  begin prev_dep(link(q)) ← prev_dep(p); link(prev_dep(p)) ← link(q); t ← type(p); type(p) ← known;
  value(p) ← value(q); free_node(q, dep_node_size);
  if abs(value(p)) ⩾ fraction_one then val_too_big(value(p));
  if internal[tracing_equations] > 0 then
    if interesting(p) then
      begin begin_diagnostic; print_nl("####␣"); print_variable_name(p); print_char("=");
      print_scaled(value(p)); end_diagnostic(false);
      end;
  if cur_exp = p then
    if cur_type = t then
      begin cur_type ← known; cur_exp ← value(p); free_node(p, value_node_size);
      end;
  end;
```

604. Процедура *fix_dependencies* выполняется, когда *fix_needed* установлена [требуется масштабировать хотябы одну независимую переменную]. Программа хранит список s независимых переменных, коэффициенты которых должны делиться на 4.

В необычных случаях, эта подгонка [масштабирование] может уменьшить один или несколько коэффициентов до нуля, так что переменная станет более или менее известной по умолчанию.

```
⟨ Объяви основные подпрограммы для списков зависимостей 594 ⟩ +≡
procedure fix_dependencies;
  label done;
  var p, q, r, s, t: pointer;  { список рабочих регистров }
    x: pointer;  { независимая переменная }
  begin r ← link(dep_head); s ← null;
  while r ≠ dep_head do
    begin t ← r; ⟨ Пройди по списку зависимостей для переменной t, исправляя все узлы, и
        заканчивая последней ссылкой q 605 ⟩;
    r ← link(q);
    if q = dep_list(t) then make_known(t, q);
    end;
  while s ≠ null do
    begin p ← link(s); x ← info(s); free_avail(s); s ← p; type(x) ← independent;
    value(x) ← value(x) + 2;
    end;
  fix_needed ← false;
  end;
```

605. **define** *independent_being_fixed* = 1 { эта переменная уже появлялась в s }

```
⟨ Пройди по списку зависимостей для переменной t, исправляя все узлы, и заканчивая последней
      ссылкой q 605 ⟩ ≡
  r ← value_loc(t);  { link(r) = dep_list(t) }
  loop begin q ← link(r); x ← info(q);
    if x = null then goto done;
    if type(x) ⩽ independent_being_fixed then
      begin if type(x) < independent_being_fixed then
        begin p ← get_avail; link(p) ← s; s ← p; info(s) ← x; type(x) ← independent_being_fixed;
        end;
      value(q) ← value(q) div 4;
      if value(q) = 0 then
        begin link(r) ← link(q); free_node(q, dep_node_size); q ← r;
        end;
      end;
    r ← q;
    end;
done:
```

Этот код используется в разделе 604.

606. Подпрограмма *new_dep* устанавливает список зависимостей p в узел зачений q, вставляя его в список всех известных значений. Мы предполагаем, что *dep_final* указывает на последний узел списка p.

```
procedure new_dep(q,p:pointer);
  var r: pointer; { что используется в качестве первой зависимости }
  begin dep_list(q)←p; prev_dep(q)←dep_head; r←link(dep_head); link(dep_final)←r;
  prev_dep(r)←dep_final; link(dep_head)←q;
  end;
```

607. Вот один из способов начать список зависимостей. Процедура *const_dependency* производит список, состоящий только из постоянного члена.

```
function const_dependency(v:scaled): pointer;
  begin dep_final←get_node(dep_node_size); value(dep_final)←v; info(dep_final)←null;
  const_dependency←dep_final;
  end;
```

608. А вот более любопытный способ начать список зависимостей с нуля: параметром для процедуры *single_dependency* является место независимой переменной x, а итогом — простой список зависимостей '$x+0$'.

В неприятном случае, когда данная независимая переменная столь часто удваивалась, что мы не можем ссылаться на неё с ненулевым коэффициентом, *single_dependency* возвращает просто список '0'. Этот случай можно распознать проверкой, что возвращаемый указатель на список равен [значению переменной] *dep_final*.

```
function single_dependency(p:pointer): pointer;
  var q: pointer; { новый список зависимостей }
    m: integer; { число удвоений }
  begin m←value(p) mod s_scale;
  if m>28 then single_dependency←const_dependency(0)
  else begin q←get_node(dep_node_size); value(q)←two_to_the[28−m]; info(q)←p;
    link(q)←const_dependency(0); single_dependency←q;
    end;
  end;
```

609. Иногда нам нужно сделать точную копию списка зависимостей.

```
function copy_dep_list(p:pointer): pointer;
  label done;
  var q: pointer; { новый список зависимостей }
  begin q←get_node(dep_node_size); dep_final←q;
  loop begin info(dep_final)←info(p); value(dep_final)←value(p);
    if info(dep_final)=null then goto done;
    link(dep_final)←get_node(dep_node_size); dep_final←link(dep_final); p←link(p);
    end;
done: copy_dep_list←q;
  end;
```

610. Но как обычно переменные становятся известными? Ах, сейчас мы дошли до сердца механизма решения уравнений. Процедуре *linear_eq* даётся список p типа *dependent* или типа *proto_dependent*, в котором есть по крайней мере одна независимая переменная. Она приравнивает этот список нулю, выбирая независимую переменную с наибольшим коэффициентом и делая её зависимой от других. Новая зависимая переменная удаляется из всех списков текущих зависимостей, таким образом, возможно, делая известными другие зависимые переменные.

Данный список p, конечно, полностью разрушается всей этой обработкой.

```
procedure linear_eq(p : pointer; t : small_number);
  var q, r, s: pointer;  { для обработки списка }
    x: pointer;  { переменная, которая теряет свою независимость }
    n: integer;  { число раз, которое x было разделено пополам }
    v: integer;  { коэффициент x в списке p }
    prev_r: pointer;  { отстаёт на один шаг от r }
    final_node: pointer;  { постоянный член нового списка зависимостей }
    w: integer;  { предполагаемый коэффициент }
  begin ⟨ Найди узел q в списке p, коэффициент v которого наибольший 611 ⟩;
  x ← info(q); n ← value(x) mod s_scale;
  ⟨ Раздели список p на −v, удаляя узел q 612 ⟩;
  if internal[tracing_equations] > 0 then ⟨ Покажи новую зависимость 613 ⟩;
  ⟨ Упрости все существующие зависимости, заменяя на x 614 ⟩;
  ⟨ Замени переменную x с independent на dependent или known 615 ⟩;
  if fix_needed then fix_dependencies;
  end;
```

611. ⟨ Найди узел q в списке p, коэффициент v которого наибольший 611 ⟩ ≡

```
  q ← p; r ← link(p); v ← value(q);
  while info(r) ≠ null do
    begin if abs(value(r)) > abs(v) then
      begin q ← r; v ← value(r);
      end;
    r ← link(r);
    end
```

Этот код используется в разделе 610.

612. Здесь мы хотим заменить коэффициенты с типа *scaled* на тип *fraction*, кроме как в постоянном члене. В общем случае тривиального уравнения вроде ‘`x=3.14`’, мы будем иметь $v = -fraction\_one$, $q = p$ и $t = dependent$.

```
⟨ Раздели список p на −v, удаляя узел q 612 ⟩ ≡
  s ← temp_head; link(s) ← p; r ← p;
  repeat if r = q then
      begin link(s) ← link(r); free_node(r, dep_node_size);
      end
    else begin w ← make_fraction(value(r), v);
      if abs(w) ⩽ half_fraction_threshold then
        begin link(s) ← link(r); free_node(r, dep_node_size);
        end
      else begin value(r) ← −w; s ← r;
        end;
      end;
    r ← link(s);
  until info(r) = null;
  if t = proto_dependent then value(r) ← −make_scaled(value(r), v)
  else if v ≠ −fraction_one then value(r) ← −make_fraction(value(r), v);
  final_node ← r; p ← link(temp_head)
```

Этот код используется в разделе 610.

613.

```
⟨ Покажи новую зависимость 613 ⟩ ≡
  if interesting(x) then
    begin begin_diagnostic; print_nl("##␣"); print_variable_name(x); w ← n;
    while w > 0 do
      begin print("*4"); w ← w − 2;
      end;
    print_char("="); print_dependency(p, dependent); end_diagnostic(false);
    end
```

Этот код используется в разделе 610.

614.

```
⟨ Упрости все существующие зависимости, заменяя на x 614 ⟩ ≡
  prev_r ← dep_head; r ← link(dep_head);
  while r ≠ dep_head do
    begin s ← dep_list(r); q ← p_with_x_becoming_q(s, x, p, type(r));
    if info(q) = null then make_known(r, q)
    else begin dep_list(r) ← q;
      repeat q ← link(q);
      until info(q) = null;
      prev_r ← q;
      end;
    r ← link(prev_r);
    end
```

Этот код используется в разделе 610.

615. ⟨ Замени переменную x с *independent* на *dependent* или *known* 615 ⟩ ≡

```
if n > 0 then ⟨ Раздели список p на 2^n 616 ⟩;
if info(p) = null then
  begin type(x) ← known; value(x) ← value(p);
  if abs(value(x)) ⩾ fraction_one then val_too_big(value(x));
  free_node(p, dep_node_size);
  if cur_exp = x then
    if cur_type = independent then
      begin cur_exp ← value(x); cur_type ← known; free_node(x, value_node_size);
      end;
  end
else begin type(x) ← dependent; dep_final ← final_node; new_dep(x, p);
  if cur_exp = x then
    if cur_type = independent then cur_type ← dependent;
  end
```

Этот код используется в разделе 610.

616. ⟨ Раздели список p на 2^n 616 ⟩ ≡

```
begin s ← temp_head; link(temp_head) ← p; r ← p;
repeat if n > 30 then w ← 0
  else w ← value(r) div two_to_the[n];
  if (abs(w) ⩽ half_fraction_threshold) ∧ (info(r) ≠ null) then
    begin link(s) ← link(r); free_node(r, dep_node_size);
    end
  else begin value(r) ← w; s ← r;
    end;
  r ← link(s);
until info(s) = null;
p ← link(temp_head);
end
```

Этот код используется в разделе 615.

617. Процедура *check_mem*, используемая только во время отладки METAFONT, проверяет, имеют ли текущие списки зависимостей правильный вид.

⟨ Проверь список линейных зависимостей 617 ⟩ ≡

```
q ← dep_head; p ← link(q);
while p ≠ dep_head do
  begin if prev_dep(p) ≠ q then
    begin print_nl("Bad␣PREVDEP␣at␣"); print_int(p);
    end;
  p ← dep_list(p); r ← inf_val;
  repeat if value(info(p)) ⩾ value(r) then
      begin print_nl("Out␣of␣order␣at␣"); print_int(p);
      end;
    r ← info(p); q ← p; p ← link(q);
  until r = null;
  end
```

Этот код используется в разделе 180.

618. Динамические нелинейные уравнения. [Dynamic nonlinear equations] Переменные численного типа [numeric type] обрабатываются по общей схеме независимых, зависимых и известных значений, которую мы только что изучили; и составляющие пар, и переменные преобразования обрабатываются таким же образом. Но METAFONT имеет ещё пять других типов значений: **boolean**, **string**, **pen**, **path** и **picture**; как насчёт них?

Уравнения допускаются между нелинейными величинами, но только в простом виде. Две переменные, которым не присвоены ещё значения, или равны друг другу, или нет.

До того, как булева переменная получила значение, её тип суть *unknown_boolean*; таким же образом, есть переменные, тип которых *unknown_string*, *unknown_pen*, *unknown_path* и *unknown_picture*. В этих случаях значение является или *null* (что означает, что нет других переменнах эквивалентных этой переменной), или оно указывает на другую переменную такого же неопределенного типа. Указатели в последнем случае образуют кольцо узлов, которое мы так и будем называть «кольцом». Кольца неопределённых переменных могут включать капсулы, возникающие как промежуточные результаты внутри выражений или как параметры **expr** макросов.

Когда один член кольца принимает значение, тогда то же самое значение даётся и всем остальным членам. В случае путей и картинок, это подразумевает создание отдельных копий, может быть, большой структуры данных; пользователям не следует увлекаться такими общностями, если только они не обладают огромным колличеством памяти.

619. Следующая процедура вызывается при добавлении узла капсулы в кольцо (например, когда неизвестная переменная упоминается в выражении).

```
function new_ring_entry(p : pointer): pointer;
  var q: pointer; { новый узел капсулы }
  begin q ← get_node(value_node_size); name_type(q) ← capsule; type(q) ← type(p);
  if value(p) = null then value(q) ← p else value(q) ← value(p);
  value(p) ← q; new_ring_entry ← q;
  end;
```

620. Наоборот, мы можем удалить капсулу или переменную перед тем, как она станет известной. Следующая процедура просто удаляет член из его кольца, не освобождая память.

```
⟨ Объяви подпрограммы для повторного использования памяти 268 ⟩ +≡
procedure ring_delete(p : pointer);
  var q: pointer;
  begin q ← value(p);
  if q ≠ null then
    if q ≠ p then
      begin while value(q) ≠ p do q ← value(q);
      value(q) ← value(p);
      end;
  end;
```

621. В конце концов, может быть уравнение, присваивающее значения всем переменным кольца. Подпрограмма *nonlinear_eq* выполняет необходимую передачу значений.

Если параметр *flush_p* имеет значение *true*, узлу p не нужно присваивать значение; он вскоре будет освобождён [recycled].

```
procedure nonlinear_eq(v : integer; p : pointer; flush_p : boolean);
  var t: small_number;   { тип кольца p }
    q,r: pointer;   { регистры для обработки ссылок }
  begin t ← type(p) − unknown_tag; q ← value(p);
  if flush_p then type(p) ← vacuous else p ← q;
  repeat r ← value(q); type(q) ← t;
    case t of
    boolean_type: value(q) ← v;
    string_type: begin value(q) ← v; add_str_ref(v);
      end;
    pen_type: begin value(q) ← v; add_pen_ref(v);
      end;
    path_type: value(q) ← copy_path(v);
    picture_type: value(q) ← copy_edges(v);
    end;   { нет больше случаев }
    q ← r;
  until q = p;
  end;
```

622. Если два члена [двух] колец приравнены, и если они имеют один и тот же тип, вызывается процедура *ring_merge*, чтобы сделать их эквивалентными.

```
procedure ring_merge(p,q : pointer);
  label exit;
  var r: pointer;   { проходит по одному списку }
  begin r ← value(p);
  while r ≠ p do
    begin if r = q then
      begin ⟨ Заяви об излишнем уравнении 623 ⟩;
      return;
      end;
    r ← value(r);
    end;
  r ← value(p); value(p) ← value(q); value(q) ← r;
exit: end;
```

623. ⟨ Заяви об излишнем уравнении 623 ⟩ ≡

```
  begin print_err("Redundant␣equation");
  help2("I␣already␣knew␣that␣this␣equation␣was␣true.")
  ("But␣perhaps␣no␣harm␣has␣been␣done;␣let's␣continue.");
  put_get_error;
  end
```

Этот код используется в разделах 622, 1004 и 1008.

658. Получение следующей лексемы. [Getting the next token] Сердце механизма ввода METAFONT'а — процедура *get_next*, которую мы будем разрабатывать в течение нескольких следующих разделов программы. Возможно, нам не стоит называть её «сердцем»: она действует как глаза и рот METAFONT'а, читая исходные файлы и пожирая их. И она также помогает METAFONT срыгивать [regurgitate] сохранённые списки лексем, которые обрабатываются вновь.

Главная обязанность [процедуры] *get_next* — ввести одну лексему и установить *cur_cmd* и *cur_mod* на код команды лексемы и её подвид. Более того, если входная лексема символьная, её *hash* адрес сохраняется в *cur_sym*, если нет, то *cur_sym* устанавливается в нуль.

За этим простым описанием скрывается определённая сложность, обусловленная всеми случаями, которые нужно обрабатывать. Однако, внутренний цикл [поцедуры] *get_next* довольно короток и быстр.

659. Перед тем, как приступить к *get_next*, нам нужно рассмотреть механизм, помогающий METAFONT'у ограничить область распространения ошибок. Всякий раз, когда программа входит в режим, где она многократно вызывает *get_next*, пока не встретится какое-либо условие, она устанавливает *scanner_status* в некоторое значение, отличное от *normal*. Когда входной файл заканчивается, или появляется символ '**outer**', возможно подходящее исправление ошибки.

Глобальная переменная *warning_info* помогает в исправлении этой ошибки, обеспечивая дополнительные сведения. Например, *warning_info* может указывать имя макроса, замещающий текст которого считывается.

```
define normal = 0  { scanner_status в «тихие времена» [“quiet times”] }
define skipping = 1  { scanner_status, когда пропускается текст не выполнившегося условия }
define flushing = 2  { scanner_status, когда хлам после оператора отбрасывается }
define absorbing = 3  { scanner_status, когда параметр text считывается }
define var_defining = 4  { scanner_status, когда vardef считывается }
define op_defining = 5  { scanner_status, когда макрос def считывается }
define loop_defining = 6  { scanner_status, когда цикл for считывается }
⟨ Общие переменные 13 ⟩ +≡
scanner_status: normal .. loop_defining;  { мы считываем на высокой скорости? }
warning_info: integer;  { если так, что ещё нам нужно знать, если случится ошибка? }
```

660. ⟨ Настрой процедуры ввода 657 ⟩ +≡

```
scanner_status ← normal;
```

661. Следующая подпрограмма вызывается, когда обнаружена внешняя [‘**outer**’] символьная лексема, или когда достигнут конец файла. Эти два случая различаются по *cur_sym*, который в конце файла равен нулю.

```
function check_outer_validity: boolean;
  var p: pointer;  { указывает на вставленный список лексем }
  begin if scanner_status = normal then check_outer_validity ← true
  else begin deletions_allowed ← false;
    ⟨ Верни внешнюю символьную лексему, чтобы её можно было повторно прочитать 662 ⟩;
    if scanner_status > skipping then
      ⟨ Скажи пользователю об утере контроля и пробуй оправиться 663 ⟩
    else begin print_err("Incomplete␣if;␣all␣text␣was␣ignored␣after␣line␣");
      print_int(warning_info);
      help3("A␣forbidden␣`outer´␣token␣occurred␣in␣skipped␣text.")
      ("This␣kind␣of␣error␣happens␣when␣you␣say␣`if...´␣and␣forget")
      ("the␣matching␣`fi´.␣I´ve␣inserted␣a␣`fi´;␣this␣might␣work.");
      if cur_sym = 0 then
        help_line[2] ← "The␣file␣ended␣while␣I␣was␣skipping␣conditional␣text.";
      cur_sym ← frozen_fi; ins_error;
      end;
    deletions_allowed ← true; check_outer_validity ← false;
    end;
  end;
```

662. ⟨ Верни внешнюю символьную лексему, чтобы её можно было повторно прочитать 662 ⟩ ≡

```
  if cur_sym ≠ 0 then
    begin p ← get_avail; info(p) ← cur_sym; back_list(p);
        { готовься вновь прочитать символьную лексему }
    end
```

Этот код используется в разделе 661.

663. ⟨ Скажи пользователю об утере контроля и пробуй оправиться 663 ⟩ ≡

```
  begin runaway;  { печатай определённое к настоящему времени }
  if cur_sym = 0 then print_err("File␣ended")
  else begin print_err("Forbidden␣token␣found");
    end;
  print("␣while␣scanning␣"); help4("I␣suspect␣you␣have␣forgotten␣an␣`enddef´,")
  ("causing␣me␣to␣read␣past␣where␣you␣wanted␣me␣to␣stop.")
  ("I´ll␣try␣to␣recover;␣but␣if␣the␣error␣is␣serious,")
  ("you´d␣better␣type␣`E´␣or␣`X´␣now␣and␣fix␣your␣file.");
  case scanner_status of
  ⟨ Заверши сообщение об ошибке, и установи cur_sym на лексему, которая может помочь оправиться
      от ошибки 664 ⟩
  end;  { нет других случаев }
  ins_error;
  end
```

Этот код используется в разделе 661.

664. Когда мы рассматриваем различные виды ошибок, уместно изменить первую строку только что выданного сообщения помощи; *help_line*[3] указывает на строку, которую можно изменить.

```
⟨ Заверши сообщение об ошибке, и установи cur_sym на лексему, которая может помочь оправиться от
      ошибки 664 ⟩ ≡
flushing: begin print("to␣the␣end␣of␣the␣statement");
  help_line[3] ← "A␣previous␣error␣seems␣to␣have␣propagated,"; cur_sym ← frozen_semicolon;
  end;
absorbing: begin print("a␣text␣argument");
  help_line[3] ← "It␣seems␣that␣a␣right␣delimiter␣was␣left␣out,";
  if warning_info = 0 then cur_sym ← frozen_end_group
  else begin cur_sym ← frozen_right_delimiter; equiv(frozen_right_delimiter) ← warning_info;
    end;
  end;
var_defining, op_defining: begin print("the␣definition␣of␣");
  if scanner_status = op_defining then slow_print(text(warning_info))
  else print_variable_name(warning_info);
  cur_sym ← frozen_end_def;
  end;
loop_defining: begin print("the␣text␣of␣a␣"); slow_print(text(warning_info)); print("␣loop");
  help_line[3] ← "I␣suspect␣you␣have␣forgotten␣an␣`endfor´,"; cur_sym ← frozen_end_for;
  end;
```

Этот код используется в разделе 663.

665. Процедура *runaway* отображает первую часть текста, встретившегося, когда METAFONT перешёл в особое состояние считывания [присвоил *scanner_status* значение отличное от *normal*], если этот текст был сохранён.

```
⟨ Объяви процедуру runaway 665 ⟩ ≡
procedure runaway;
  begin if scanner_status > flushing then
    begin print_nl("Runaway␣");
    case scanner_status of
    absorbing: print("text?");
    var_defining, op_defining: print("definition?");
    loop_defining: print("loop?");
    end; { нет других случаев }
    print_ln; show_token_list(link(hold_head), null, error_line − 10, 0);
    end;
  end;
```

Этот код используется в разделе 162.

666. Нам нужно упомянуть процедуру, которая может вызываться процедурой *get_next*.

```
procedure firm_up_the_line; forward;
```

667. А теперь мы готовы погрузиться в саму *get_next*.

```
define switch = 25  { метка в get_next }
define start_numeric_token = 85  { ещё одна }
define start_decimal_token = 86  { и ещё одна }
define fin_numeric_token = 87  { и ещё одна, хотя goto и считается вредным }
procedure get_next;  { устанавливает cur_cmd, cur_mod, cur_sym на следующую лексему }
  label restart,  { идти сюда, чтобы получить следующую входную лексему }
    exit,  { идти сюда, когда следующая входная лексема получена }
    found,  { идти сюда, когда обнаружен конец символьной лексемы }
    switch,  { идти сюда, при ветвлении по классу входного символа }
    start_numeric_token, start_decimal_token, fin_numeric_token, done;
        { идти сюда в особых случаях, когда считывается число }
  var k: 0 .. buf_size;  { индекс в buffer }
    c: ASCII_code;  { текущий символ в буфере [buffer] }
    class: ASCII_code;  { номер его класса }
    n, f: integer;  { регистры для преобразования десятичного числа в двоичное }
  begin restart: cur_sym ← 0;
  if file_state then ⟨ Введи из внешнего файла; goto restart, если вводить нечего, или return, если
        несимвольная лексема найдена 669 ⟩
  else ⟨ Введи из списка лексем; goto restart, если конец списка, или если параметр нужно раскрыть,
        или return, если несимвольная лексема найдена 676 ⟩;
  ⟨ Заверши получение символьной лексемы в cur_sym; goto restart, если она недопустима 668 ⟩;
exit: end;
```

668. Когда символьная лексема объявляется как '**outer**', её код команды увеличивается на *outer_tag*.

```
⟨ Заверши получение символьной лексемы в cur_sym; goto restart, если она недопустима 668 ⟩ ≡
  cur_cmd ← eq_type(cur_sym); cur_mod ← equiv(cur_sym);
  if cur_cmd ⩾ outer_tag then
    if check_outer_validity then cur_cmd ← cur_cmd − outer_tag
    else goto restart
```

Этот код используется в разделе 667.

669. Знак процента появляется в *buffer*[*limit*]; это делает ненужной проверку конца строки.

```
⟨ Введи из внешнего файла; goto restart, если вводить нечего, или return, если несимвольная лексема
      найдена 669 ⟩ ≡
  begin switch: c ← buffer[loc]; incr(loc); class ← char_class[c];
  case class of
  digit_class: goto start_numeric_token;
  period_class: begin class ← char_class[buffer[loc]];
    if class > period_class then goto switch
    else if class < period_class then { class = digit_class }
        begin n ← 0; goto start_decimal_token;
        end;
    end;
  space_class: goto switch;
  percent_class: begin ⟨ Иди к следующей строке файла или goto restart, если нет следующей
        строки 679 ⟩;
    check_interrupt; goto switch;
    end;
  string_class: ⟨ Получи строковую лексему и return 671 ⟩;
  isolated_classes: begin k ← loc − 1; goto found;
    end;
  invalid_class: ⟨ Поругай недопустимый символ и goto restart 670 ⟩;
  othercases do_nothing { буквы, и т. д. }
  endcases;
  k ← loc − 1;
  while char_class[buffer[loc]] = class do incr(loc);
  goto found;
start_numeric_token: ⟨ Получи целую часть n числовой лексемы; установи f ← 0 и goto
      fin_numeric_token, если нет десятичной точки 673 ⟩;
start_decimal_token: ⟨ Получи дробную часть f числовой лексемы 674 ⟩;
fin_numeric_token: ⟨ Собери числовую и дробную части числовой лексемы и return 675 ⟩;
found: cur_sym ← id_lookup(k, loc − k);
  end
```

Этот код используется в разделе 667.

670. Мы идём к *restart* вместо *switch*, потому что *state* должно быть равно *token_list* после того, как его обработала процедура *error* (сравни с *clear_for_error_prompt*).

```
⟨ Поругай недопустимый символ и goto restart 670 ⟩ ≡
  begin print_err("Text␣line␣contains␣an␣invalid␣character");
  help2("A␣funny␣symbol␣that␣I␣can´t␣read␣has␣just␣been␣input.")
  ("Continue,␣and␣I´ll␣forget␣that␣it␣ever␣happened.");
  deletions_allowed ← false; error; deletions_allowed ← true; goto restart;
  end
```

Этот код используется в разделе 669.

671. ⟨ Получи строковую лексему и **return** 671 ⟩ ≡

```
begin if buffer[loc] = """" then cur_mod ← ""
else begin k ← loc; buffer[limit + 1] ← """";
  repeat incr(loc);
  until buffer[loc] = """";
  if loc > limit then ⟨ Поругай пропущенный разделитель строки и goto restart 672 ⟩;
  if (loc = k + 1) ∧ (length(buffer[k]) = 1) then cur_mod ← buffer[k]
  else begin str_room(loc − k);
    repeat append_char(buffer[k]); incr(k);
    until k = loc;
    cur_mod ← make_string;
    end;
  end;
incr(loc); cur_cmd ← string_token; return;
end
```

Этот код используется в разделе 669.

672. Мы идём к *restart* после этого сообщения об ошибке, а не к *switch*, потому что подпрограмма *clear_for_error_prompt* должна переустановить *token_state* после завершения *error*.

⟨ Поругай пропущенный разделитель строки и **goto** *restart* 672 ⟩ ≡

```
begin loc ← limit; { следующий знак, который будет прочитан с этой строки, суть "%" }
print_err("Incomplete␣string␣token␣has␣been␣flushed");
help3("Strings␣should␣finish␣on␣the␣same␣line␣as␣they␣began.")
("I´ve␣deleted␣the␣partial␣string;␣you␣might␣want␣to")
("insert␣another␣by␣typing,␣e.g.,␣`I""new␣string""´.");
deletions_allowed ← false; error; deletions_allowed ← true; goto restart;
end
```

Этот код используется в разделе 671.

673. ⟨ Получи целую часть n числовой лексемы; установи $f \leftarrow 0$ и **goto** *fin_numeric_token*, если нет десятичной точки 673 ⟩ ≡

```
n ← c − "0";
while char_class[buffer[loc]] = digit_class do
  begin if n < 4096 then n ← 10 * n + buffer[loc] − "0";
  incr(loc);
  end;
if buffer[loc] = "." then
  if char_class[buffer[loc + 1]] = digit_class then goto done;
f ← 0; goto fin_numeric_token;
done: incr(loc)
```

Этот код используется в разделе 669.

674. ⟨ Получи дробную часть f числовой лексемы 674 ⟩ ≡

```
k ← 0;
repeat if k < 17 then { цифры для k ⩾ 17 не могут влиять на итог }
    begin dig[k] ← buffer[loc] − "0"; incr(k);
    end;
  incr(loc);
until char_class[buffer[loc]] ≠ digit_class;
f ← round_decimals(k);
if f = unity then
  begin incr(n); f ← 0;
  end
```

Этот код используется в разделе 669.

675. ⟨ Собери числовую и дробную части числовой лексемы и **return** 675 ⟩ ≡

```
if n < 4096 then cur_mod ← n * unity + f
else begin print_err("Enormous␣number␣has␣been␣reduced");
  help2("I␣can't␣handle␣numbers␣bigger␣than␣about␣4095.99998;")
  ("so␣I've␣changed␣your␣constant␣to␣that␣maximum␣amount.");
  deletions_allowed ← false; error; deletions_allowed ← true; cur_mod ← 7777777777;
  end;
cur_cmd ← numeric_token; return
```

Этот код используется в разделе 669.

676. Давайте, сейчас рассмотрим, что происходит, когда *get_next* просматривает список лексем.

⟨ Введи из списка лексем; **goto** *restart*, если конец списка, или если параметр нужно раскрыть, или **return**, если несимвольная лексема найдена 676 ⟩ ≡

```
if loc ⩾ hi_mem_min then { однословная лексема }
  begin cur_sym ← info(loc); loc ← link(loc); { переместиться к следующей }
  if cur_sym ⩾ expr_base then
    if cur_sym ⩾ suffix_base then ⟨ Вставь суффиксный или текстовый параметр и goto restart 677 ⟩
    else begin cur_cmd ← capsule_token;
      cur_mod ← param_stack[param_start + cur_sym − (expr_base)]; cur_sym ← 0; return;
      end;
  end
else if loc > null then
    ⟨ Получи сохранённую числовую или строковую лексему, или лексему капсулы и return 678 ⟩
  else begin { заканчиваем с этим списком лексем }
    end_token_list; goto restart; { вернёмся на предыдущий уровень }
    end
```

Этот код используется в разделе 667.

677. ⟨ Вставь суффиксный или текстовый параметр и **goto** *restart* 677 ⟩ ≡

```
begin if cur_sym ⩾ text_base then cur_sym ← cur_sym − param_size;
      { param_size = text_base − suffix_base }
begin_token_list(param_stack[param_start + cur_sym − (suffix_base)], parameter); goto restart;
end
```

Этот код используется в разделе 676.

678. ⟨ Получи сохранённую числовую или строковую лексему, или лексему капсулы и **return** 678 ⟩ ≡

```
begin if name_type(loc) = token then
  begin cur_mod ← value(loc);
  if type(loc) = known then cur_cmd ← numeric_token
  else begin cur_cmd ← string_token; add_str_ref(cur_mod);
    end;
  end
else begin cur_mod ← loc; cur_cmd ← capsule_token;
  end;
loc ← link(loc); return;
end
```

Этот код используется в разделе 676.

679. Все простые ветви процедуры *get_next* разработаны. Осталась ещё одна ветвь.

⟨ Иди к следующей строке файла или **goto** *restart*, если нет следующей строки 679 ⟩ ≡

```
if name > 2 then
  ⟨ Читай следующую строку файла в buffer или goto restart, если файл закончился 681 ⟩
else begin if input_ptr > 0 then
        { текст вставлен при исправлении ошибки или командой scantokens }
    begin end_file_reading; goto restart; { возобнови предыдущий уровень }
    end;
  if selector < log_only then open_log_file;
  if interaction > nonstop_mode then
    begin if limit = start then { предыдущая строка была пустой }
      print_nl("(Please␣type␣a␣command␣or␣say␣`end´)");
    print_ln; first ← start; prompt_input("*"); { ввод строки в buffer в диалоговом режиме }
    limit ← last; buffer[limit] ← "%"; first ← limit + 1; loc ← start;
    end
  else fatal_error("***␣(job␣aborted,␣no␣legal␣end␣found)");
        { режим без остановки, предназначенный для ночной пакетной обработки, ни когда не
        ждёт ввода от оператора }
  end
```

Этот код используется в разделе 669.

680. Глобальная переменная *force_eof* обычно имеет значение *false*; она устанавливается в *true* командой **endinput**.

⟨ Общие переменные 13 ⟩ +≡

```
force_eof: boolean; { должен ли следующий input прерваться раньше? }
```

681. ⟨ Читай следующую строку файла в *buffer* или **goto** *restart*, если файл закончился 681 ⟩ ≡

```
begin incr(line); first ← start;
if ¬force_eof then
  begin if input_ln(cur_file, true) then   { не конец файла }
    firm_up_the_line   { это устанавливает limit }
  else force_eof ← true;
  end;
if force_eof then
  begin print_char(")"); decr(open_parens); update_terminal;
      { показать пользователю, что файл прочтён }
  force_eof ← false; end_file_reading;   { возобновить предыдущий уровень }
  if check_outer_validity then goto restart else goto restart;
  end;
buffer[limit] ← "%"; first ← limit + 1; loc ← start;   { готов читать }
end
```

Этот код используется в разделе 679.

682. Если пользователь установит параметр *pausing* в некоторое положительное значение, и если не был выбран режим работы без остановки [nonstop], каждая строка ввода отображается на терминале, и в файле протокола за [знаками] '=>'. METAFONT ждёт ответ. Если ответ пустой [null] (то есть, если ни чего не напечатано, кроме, возможно, нескольких пробелов), принимается исходная строка; иначе напечатанная строка используется взамен строки файла.

```
procedure firm_up_the_line;
  var k: 0 .. buf_size;   { индекс в buffer }
  begin limit ← last;
  if internal[pausing] > 0 then
    if interaction > nonstop_mode then
      begin wake_up_terminal; print_ln;
      if start < limit then
        for k ← start to limit − 1 do print(buffer[k]);
      first ← limit; prompt_input("=>");   { ждать ответ от пользователя }
      if last > first then
        begin for k ← first to last − 1 do   { переместить строку вниз в буфере }
          buffer[k + start − first] ← buffer[k];
        limit ← start + last − first;
        end;
      end;
  end;
```

683. Считывание макроопределений. [Scanning macro definitions] У METAFONT'а много способов помещать лексемы в списки лексем для последующего использования: макросы можно определять с помощью **def**, **vardef**, **primarydef** и т. д.; повторяющийся код можно определить с помощью **for**, **forever**, **forsuffixes**. Все эти действия выполняются подпрограммами из этой части программы.

Изменяемая часть [подвид] каждого кода команды равна нулю для «завершающих ограничителей» вроде **enddef** и **endfor**.

```
define start_def = 1  { подвид команды для def }
define var_def = 2  { подвид команды для vardef }
define end_def = 0  { подвид команды для enddef }
define start_forever = 1  { подвид команды для forever }
define end_for = 0  { подвид команды для endfor }
⟨ Помести каждый примитив METAFONT'а в хеш-таблицу 192 ⟩ +≡
  primitive("def", macro_def, start_def);
  primitive("vardef", macro_def, var_def);
  primitive("primarydef", macro_def, secondary_primary_macro);
  primitive("secondarydef", macro_def, tertiary_secondary_macro);
  primitive("tertiarydef", macro_def, expression_tertiary_macro);
  primitive("enddef", macro_def, end_def); eqtb[frozen_end_def] ← eqtb[cur_sym];
  primitive("for", iteration, expr_base);
  primitive("forsuffixes", iteration, suffix_base);
  primitive("forever", iteration, start_forever);
  primitive("endfor", iteration, end_for); eqtb[frozen_end_for] ← eqtb[cur_sym];
```

684. ⟨ Случаи *print_cmd_mod* для символьной печати примитивов 212 ⟩ +≡

```
macro_def: if m ⩽ var_def then
    if m = start_def then print("def")
    else if m < start_def then print("enddef")
      else print("vardef")
  else if m = secondary_primary_macro then print("primarydef")
    else if m = tertiary_secondary_macro then print("secondarydef")
      else print("tertiarydef");
iteration: if m ⩽ start_forever then
    if m = start_forever then print("forever") else print("endfor")
  else if m = expr_base then print("for") else print("forsuffixes");
```

685. Различные действия, поглащающие макросы, имеют различный синтаксис, но они также имеют и много общего. Есть список особых символов, которые не должны замещаться лексемами параметров; есть особый код команды, завершающий определение; приведённые соглашения тождественны. Поэтому имеет смысл выполнять бóльшую часть работы одной подпрограммой. Эта подпрограмма называется *scan_toks*.

Первый параметр для *scan_toks* — код команды, завершающей считывание (*macro_def*, *loop_repeat* или *iteration*).

Второй параметр *subst_list* указывает на (возможно пустой) список двусловных узлов, поля которых *info* и *value* указывают символьные лексемы до и после замещения. Список возвратится в свободное хранилище [free storage] функцией *scan_toks*.

Третий параметр просто добавляется к создаваемому списку лексем. И последний параметр указывает, сколько особых действий `#@`, `@` и `@#` нужно заменить параметрами суффиксов. Если такие параметры есть, они называются `(SUFFIX0)`, `(SUFFIX1)` и `(SUFFIX2)`.

```
function scan_toks(terminator : command_code; subst_list, tail_end : pointer; suffix_count : small_number):
        pointer;
  label done, found;
  var p: pointer;  { конец создаваемого списка лексем }
    q: pointer;  { временная переменная для управления ссылками }
    balance: integer;  { левые разделители минус правые разделители }
  begin p ← hold_head; balance ← 1; link(hold_head) ← null;
  loop begin get_next;
    if cur_sym > 0 then
      begin ⟨ Замени cur_sym, если он в subst_list 686 ⟩;
      if cur_cmd = terminator then ⟨ Подправь баланс; если он нулевой, goto done 687 ⟩
      else if cur_cmd = macro_special then ⟨ Обработай знаки в кавычках: #@, @ или @# 690 ⟩;
      end;
    link(p) ← cur_tok; p ← link(p);
    end;
done: link(p) ← tail_end; flush_node_list(subst_list); scan_toks ← link(hold_head);
  end;
```

686. ⟨ Замени *cur_sym*, если он в *subst_list* 686 ⟩ ≡

```
  begin q ← subst_list;
  while q ≠ null do
    begin if info(q) = cur_sym then
      begin cur_sym ← value(q); cur_cmd ← relax; goto found;
      end;
    q ← link(q);
    end;
found: end
```

Этот код используется в разделе 685.

687. ⟨ Подправь баланс; если он нулевой, **goto** *done* 687 ⟩ ≡

```
  if cur_mod > 0 then incr(balance)
  else begin decr(balance);
    if balance = 0 then goto done;
    end
```

Этот код используется в разделе 685.

688. Четыре команды предназначены для использования только внутри текстов макросов: **quote**, `#@`, `@` и `@#`. Все они суть подвиды одного и того же кода команды *macro_special*.

```
define quote = 0   { macro_special подвид для quote }
define macro_prefix = 1   { macro_special подвид для #@ }
define macro_at = 2   { macro_special подвид для @ }
define macro_suffix = 3   { macro_special подвид для @# }
```

⟨ Помести каждый примитив METAFONT'а в хеш-таблицу 192 ⟩ +≡

```
primitive("quote", macro_special, quote);
primitive("#@", macro_special, macro_prefix);
primitive("@", macro_special, macro_at);
primitive("@#", macro_special, macro_suffix);
```

689. ⟨ Случаи *print_cmd_mod* для символьной печати примитивов 212 ⟩ +≡

```
macro_special: case m of
  macro_prefix: print("#@");
  macro_at: print_char("@");
  macro_suffix: print("@#");
  othercases print("quote")
  endcases;
```

690. ⟨ Обработай знаки в кавычках: `#@`, `@` или `@#` 690 ⟩ ≡

```
begin if cur_mod = quote then get_next
else if cur_mod ⩽ suffix_count then cur_sym ← suffix_base − 1 + cur_mod;
end
```

Этот код используется в разделе 685.

691. Вот подпрограмма, используемая при каждом переопределении лексемы. Если лексема пользователя непереопределяема, подставляется лексема '*frozen_inaccessible*'; её можно переопределить, но нельзя использовать, поэтому таблицы METAFONT'а не портятся.

```
procedure get_symbol; { устанавливает cur_sym на безопасный символ }
  label restart;
  begin restart: get_next;
  if (cur_sym = 0) ∨ (cur_sym > frozen_inaccessible) then
    begin print_err("Missing␣symbolic␣token␣inserted");
    help3("Sorry:␣You␣can´t␣redefine␣a␣number,␣string,␣or␣expr.")
    ("I´ve␣inserted␣an␣inaccessible␣symbol␣so␣that␣your")
    ("definition␣will␣be␣completed␣without␣mixing␣me␣up␣too␣badly.");
    if cur_sym > 0 then help_line[2] ← "Sorry:␣You␣can´t␣redefine␣my␣error-recovery␣tokens."
    else if cur_cmd = string_token then delete_str_ref(cur_mod);
    cur_sym ← frozen_inaccessible; ins_error; goto restart;
    end;
  end;
```

692. Перед тем, как мы переопределим символьную лексему, нам нужно убрать её предыдущее значение, если она была переменной. Более строгая разновидность процедуры *get_symbol* делает это.

```
procedure get_clear_symbol;
  begin get_symbol; clear_symbol(cur_sym, false);
  end;
```

693. Вот другая маленькая подпрограмма, она проверяет, идёт ли знак равенства или присвоения в надлежащем месте в определении макроса.

```
procedure check_equals;
  begin if cur_cmd ≠ equals then
    if cur_cmd ≠ assignment then
      begin missing_err("=");
      help5("The next thing in this `def´ should have been `=´,")
      ("because I´ve already looked at the definition heading.")
      ("But don´t worry; I´ll pretend that an equals sign")
      ("was present. Everything from here to `enddef´")
      ("will be the replacement text of this macro."); back_error;
      end;
  end;
```

694. Команды **primarydef**, **secondarydef** или **tertiarydef** довольно легко выполняются теперь, когда мы имеем *scan_toks*. В этом случае есть два параметра, которые будут `EXPR0` и `EXPR1` (то есть, *expr_base* и *expr_base* + 1).

```
procedure make_op_def;
  var m: command_code; { вид определения }
    p, q, r: pointer; { переменные для работы со списком }
  begin m ← cur_mod;
  get_symbol; q ← get_node(token_node_size); info(q) ← cur_sym; value(q) ← expr_base;
  get_clear_symbol; warning_info ← cur_sym;
  get_symbol; p ← get_node(token_node_size); info(p) ← cur_sym; value(p) ← expr_base + 1; link(p) ← q;
  get_next; check_equals;
  scanner_status ← op_defining; q ← get_avail; ref_count(q) ← null; r ← get_avail; link(q) ← r;
  info(r) ← general_macro; link(r) ← scan_toks(macro_def, p, null, 0); scanner_status ← normal;
  eq_type(warning_info) ← m; equiv(warning_info) ← q; get_x_next;
  end;
```

695. Параметры для макросов вводятся ключевыми словами **expr**, **suffix**, **text**, **primary**, **secondary** и **tertiary**.

```
⟨ Помести каждый примитив METAFONT'а в хеш-таблицу 192 ⟩ +≡
  primitive("expr", param_type, expr_base);
  primitive("suffix", param_type, suffix_base);
  primitive("text", param_type, text_base);
  primitive("primary", param_type, primary_macro);
  primitive("secondary", param_type, secondary_macro);
  primitive("tertiary", param_type, tertiary_macro);
```

696. ⟨ Случаи *print_cmd_mod* для символьной печати примитивов 212 ⟩ +≡

```
param_type: if m ⩾ expr_base then
    if m = expr_base then print("expr")
    else if m = suffix_base then print("suffix")
      else print("text")
  else if m < secondary_macro then print("primary")
    else if m = secondary_macro then print("secondary")
      else print("tertiary");
```

697. Обратимся к более сложной обработке, связанной с **def** и **vardef**. Когда вызывается следующая процедура, значение *cur_mod* должно быть или *start_def*, или *var_def*.

```
⟨ Объяви процедуру check_delimiter 1032 ⟩
⟨ Объяви функцию scan_declared_variable 1011 ⟩
procedure scan_def;
  var m: start_def .. var_def;  { тип определения }
    n: 0 .. 3;  { число особых суффиксных параметров }
    k: 0 .. param_size;  { общее число параметров }
    c: general_macro .. text_macro;  { вид определяемого макроса }
    r: pointer;  { список подстановок параметров }
    q: pointer;  { остаток списка лексем макроса }
    p: pointer;  { временная переменная }
    base: halfword;  { expr_base, suffix_base или text_base }
    l_delim, r_delim: pointer;  { соответствующие разделителей }
  begin m ← cur_mod; c ← general_macro; link(hold_head) ← null;
  q ← get_avail; ref_count(q) ← null; r ← null;
  ⟨ Считай лексему или переменную, чтобы определить; установи n, scanner_status и warning_info 700 ⟩;
  k ← n;
  if cur_cmd = left_delimiter then
    ⟨ Собери разграниченные параметры, помещая их в списки q и r 703 ⟩;
  if cur_cmd = param_type then ⟨ Собери неразграниченные параметры, помещая их в список r 705 ⟩;
  check_equals; p ← get_avail; info(p) ← c; link(q) ← p;
  ⟨ Присоедини замещаемый текст к концу узла p 698 ⟩;
  scanner_status ← normal; get_x_next;
  end;
```

698. Мы не помещаем '*frozen_end_group*' в замещающий текст [команды] **vardef**, потому что пользователь может захотеть переопределить '`endgroup`'.

```
⟨ Присоедини замещаемый текст к концу узла p 698 ⟩ ≡
  if m = start_def then link(p) ← scan_toks(macro_def, r, null, n)
  else begin q ← get_avail; info(q) ← bg_loc; link(p) ← q; p ← get_avail; info(p) ← eg_loc;
    link(q) ← scan_toks(macro_def, r, p, n);
    end;
  if warning_info = bad_vardef then flush_token_list(value(bad_vardef))
```

Этот код используется в разделе 697.

699.
```
⟨ Общие переменные 13 ⟩ +≡
bg_loc, eg_loc: 1 .. hash_end;  { хеш-адрес команд 'begingroup' и 'endgroup' }
```

700. ⟨ Считай лексему или переменную, чтобы определить; установи n, *scanner_status* и *warning_info* 700 ⟩ ≡

```
if m = start_def then
  begin get_clear_symbol; warning_info ← cur_sym; get_next; scanner_status ← op_defining; n ← 0;
  eq_type(warning_info) ← defined_macro; equiv(warning_info) ← q;
  end
else begin p ← scan_declared_variable; flush_variable(equiv(info(p)), link(p), true);
  warning_info ← find_variable(p); flush_list(p);
  if warning_info = null then ⟨ Замени на 'плохую переменную' 701 ⟩;
  scanner_status ← var_defining; n ← 2;
  if cur_cmd = macro_special then
    if cur_mod = macro_suffix then {@#}
      begin n ← 3; get_next;
      end;
  type(warning_info) ← unsuffixed_macro − 2 + n; value(warning_info) ← q;
  end {suffixed_macro = unsuffixed_macro + 1}
```

Этот код используется в разделе 697.

701. ⟨ Замени на 'плохую переменную' 701 ⟩ ≡

```
begin print_err("This␣variable␣already␣starts␣with␣a␣macro");
help2("After␣`vardef␣a´␣you␣can´t␣say␣`vardef␣a.b´.")
("So␣I´ll␣have␣to␣discard␣this␣definition."); error; warning_info ← bad_vardef;
end
```

Этот код используется в разделе 700.

702. ⟨ Настрой таблицы (делает только INIMF) 176 ⟩ +≡

```
name_type(bad_vardef) ← root; link(bad_vardef) ← frozen_bad_vardef;
equiv(frozen_bad_vardef) ← bad_vardef; eq_type(frozen_bad_vardef) ← tag_token;
```

703. ⟨ Собери разграниченные параметры, помещая их в списки q и r 703 ⟩ ≡

```
repeat l_delim ← cur_sym; r_delim ← cur_mod; get_next;
  if (cur_cmd = param_type) ∧ (cur_mod ⩾ expr_base) then base ← cur_mod
  else begin print_err("Missing␣parameter␣type;␣`expr´␣will␣be␣assumed");
    help1("You␣should´ve␣had␣`expr´␣or␣`suffix´␣or␣`text´␣here."); back_error;
    base ← expr_base;
    end;
  ⟨ Собери лексемы параметров для типа base 704 ⟩;
  check_delimiter(l_delim, r_delim); get_next;
until cur_cmd ≠ left_delimiter
```

Этот код используется в разделе 697.

704. ⟨ Собери лексемы параметров для типа *base* 704 ⟩ ≡

```
repeat link(q) ← get_avail; q ← link(q); info(q) ← base + k;
  get_symbol; p ← get_node(token_node_size); value(p) ← base + k; info(p) ← cur_sym;
  if k = param_size then overflow("parameter␣stack␣size", param_size);
  incr(k); link(p) ← r; r ← p; get_next;
until cur_cmd ≠ comma
```

Этот код используется в разделе 703.

705. ⟨ Собери неразграниченные параметры, помещая их в список r 705 ⟩ ≡

```
begin p ← get_node(token_node_size);
if cur_mod < expr_base then
  begin c ← cur_mod; value(p) ← expr_base + k;
  end
else begin value(p) ← cur_mod + k;
  if cur_mod = expr_base then c ← expr_macro
  else if cur_mod = suffix_base then c ← suffix_macro
    else c ← text_macro;
  end;
if k = param_size then overflow("parameter␣stack␣size", param_size);
incr(k); get_symbol; info(p) ← cur_sym; link(p) ← r; r ← p; get_next;
if c = expr_macro then
  if cur_cmd = of_token then
    begin c ← of_macro; p ← get_node(token_node_size);
    if k = param_size then overflow("parameter␣stack␣size", param_size);
    value(p) ← expr_base + k; get_symbol; info(p) ← cur_sym; link(p) ← r; r ← p; get_next;
    end;
end
```

Этот код используется в разделе 697.

706. Раскрытие следующей лексемы. [Expanding the next token] Лишь несколько кодов команд меньших, чем *min_command*, может возвращаться функцией *get_next*; в возрастающем порядке это *if_test*, *fi_or_else*, *input*, *iteration*, *repeat_loop*, *exit_test*, *relax*, *scan_tokens*, *expand_after* и *defined_macro*.

METAFONT обычно получает следующую лексему ввода, вызывая *get_x_next*. Она похожа на *get_next*, но получает больше лексем, пока не находит $cur\_cmd \geqslant min\_command$. Другими словами, *get_x_next* раскрывает макросы и удаляет встречающиеся условия, повторения и команды ввода.

Отсюда вытекает, что *get_x_next* может вызывать сама себя рекурсивно. По сути, есть массив рекурсий, т. к. раскрытие макросов может вызывать считывание произвольно сложных выражений, которые по очереди вызывают раскрытие макросов и условий, и т. д.

Поэтому необходимо объявить все ветви *forward* процедур в этом месте, и вставить некоторые другие процедуры, которые будут вызываться процедурой *get_x_next*.

```
procedure scan_primary; forward;
procedure scan_secondary; forward;
procedure scan_tertiary; forward;
procedure scan_expression; forward;
procedure scan_suffix; forward;
⟨ Объяви процедуру macro_call 720 ⟩
procedure get_boolean; forward;
procedure pass_text; forward;
procedure conditional; forward;
procedure start_input; forward;
procedure begin_iteration; forward;
procedure resume_iteration; forward;
procedure stop_iteration; forward;
```

707. Вспомогательная подпрограмма *expand* используется процедурой *get_x_next*, когда она должна делать необычные команды раскрытия.

```
procedure expand;
  var p: pointer; { для работы со списком }
    k: integer; { нечто ⩽ buf_size, как мы надеемся }
    j: pool_pointer; { индекс в str_pool }
  begin if internal[tracing_commands] > unity then
    if cur_cmd ≠ defined_macro then show_cur_cmd_mod;
  case cur_cmd of
  if_test: conditional; { эта процедура обсуждается в Части 36 ниже }
  fi_or_else: ⟨ Заверши текущее условие и пропусти до fi 751 ⟩;
  input: ⟨ Начни или заверши ввод из файла 711 ⟩;
  iteration: if cur_mod = end_for then ⟨ Поругай пользователя за лишнее endfor 708 ⟩
    else begin_iteration; { эта процедура обсуждается в Части 37 ниже }
  repeat_loop: ⟨ Повторяй цикл 712 ⟩;
  exit_test: ⟨ Выйди из цикла, если пришло время 713 ⟩;
  relax: do_nothing;
  expand_after: ⟨ Раскрой эту лексему после следующей 715 ⟩;
  scan_tokens: ⟨ Помести строку во входной буфер 716 ⟩;
  defined_macro: macro_call(cur_mod, null, cur_sym);
  end; { нет других случаев }
  end;
```

708. ⟨ Поругай пользователя за лишнее **endfor** 708 ⟩ ≡

```
begin print_err("Extra␣`endfor´");  help2("I´m␣not␣currently␣working␣on␣a␣for␣loop,")
("so␣I␣had␣better␣not␣try␣to␣end␣anything.");
error;
end
```

Этот код используется в разделе 707.

709. Обработка **input** вызывает процедуру *start_input*, которая будет объявлена позже, обработка **endinput** очевидна.

⟨ Помести каждый примитив METAFONT'а в хеш-таблицу 192 ⟩ +≡

```
primitive("input", input, 0);
primitive("endinput", input, 1);
```

710. ⟨ Случаи *print_cmd_mod* для символьной печати примитивов 212 ⟩ +≡

```
input: if m = 0 then print("input") else print("endinput");
```

711. ⟨ Начни или заверши ввод из файла 711 ⟩ ≡

```
if cur_mod > 0 then force_eof ← true
else start_input
```

Этот код используется в разделе 707.

712. Мы обсудим сложные части действий с циклами позже. Сейчас достаточно знать, что есть глобальная переменная *loop_ptr*, которая будет равна *null*, если цикл не обрабатывается.

⟨ Повторяй цикл 712 ⟩ ≡

```
begin while token_state ∧ (loc = null) do end_token_list;  { сохраняет место в стеке }
if loop_ptr = null then
  begin print_err("Lost␣loop");
  help2("I´m␣confused;␣after␣exiting␣from␣a␣loop,␣I␣still␣seem")
  ("to␣want␣to␣repeat␣it.␣I´ll␣try␣to␣forget␣the␣problem.");
  error;
  end
else resume_iteration;  { эта процедура в Части 37 ниже }
end
```

Этот код используется в разделе 707.

713. ⟨ Выйди из цикла, если пришло время 713 ⟩ ≡

```
begin get_boolean;
if internal[tracing_commands] > unity then show_cmd_mod(nullary, cur_exp);
if cur_exp = true_code then
  if loop_ptr = null then
    begin print_err("No loop is in progress");
    help1("Why say `exitif' when there's nothing to exit from?");
    if cur_cmd = semicolon then error else back_error;
    end
  else ⟨ Выйди раньше времени из итерации 714 ⟩
else if cur_cmd ≠ semicolon then
    begin missing_err(";");
    help2("After `exitif <boolean exp>' I expect to see a semicolon.")
    ("I shall pretend that one was there."); back_error;
    end;
end
```

Этот код используется в разделе 707.

714. Здесь мы пользуемся тем, что *forever_text* — это просто *token_type*, значение которой меньше *loop_text*.

⟨ Выйди раньше времени из итерации 714 ⟩ ≡

```
begin p ← null;
repeat if file_state then end_file_reading
  else begin if token_type ⩽ loop_text then p ← start;
    end_token_list;
    end;
until p ≠ null;
if p ≠ info(loop_ptr) then fatal_error("*** (loop confusion)");
stop_iteration; { эта процедура в Части 37 ниже }
end
```

Этот код используется в разделе 713.

715. ⟨ Раскрой эту лексему после следующей 715 ⟩ ≡

```
begin get_next; p ← cur_tok; get_next;
if cur_cmd < min_command then expand
else back_input;
back_list(p);
end
```

Этот код используется в разделе 707.

716. ⟨ Помести строку во входной буфер 716 ⟩ ≡

```
begin get_x_next; scan_primary;
if cur_type ≠ string_type then
  begin disp_err(null, "Not a string"); help2("I'm going to flush this expression, since")
  ("scantokens should be followed by a known string."); put_get_flush_error(0);
  end
else begin back_input;
  if length(cur_exp) > 0 then ⟨ Притворись, что мы читаем новый однострочный файл 717 ⟩;
  end;
end
```

Этот код используется в разделе 707.

717. ⟨ Притворись, что мы читаем новый однострочный файл 717 ⟩ ≡

```
begin begin_file_reading; name ← 2; k ← first + length(cur_exp);
if k ⩾ max_buf_stack then
  begin if k ⩾ buf_size then
    begin max_buf_stack ← buf_size; overflow("buffer␣size", buf_size);
    end;
  max_buf_stack ← k + 1;
  end;
j ← str_start[cur_exp]; limit ← k;
while first < limit do
  begin buffer[first] ← so(str_pool[j]); incr(j); incr(first);
  end;
buffer[limit] ← "%"; first ← limit + 1; loc ← start; flush_cur_exp(0);
end
```

Этот код используется в разделе 716.

718. Здесь, наконец *get_x_next*.

Подпрограммы считывания выражений, рассматриваемые позднее, общаются через глобальные величины *cur_type* и *cur_exp*; мы должны быть очень осторожны с сохранением и восстановлением этих величин во время раскрытия макросов.

```
procedure get_x_next;
  var save_exp: pointer;  { капсула для хранения cur_type и cur_exp }
  begin get_next;
  if cur_cmd < min_command then
    begin save_exp ← stash_cur_exp;
    repeat if cur_cmd = defined_macro then macro_call(cur_mod, null, cur_sym)
      else expand;
      get_next;
    until cur_cmd ⩾ min_command;
    unstash_cur_exp(save_exp);  { то, что хранит cur_type и cur_exp }
    end;
  end;
```

719. Сейчас, давайте, рассмотрим процедуру *macro_call*, которая используется для начинала работы всех определённых пользователем макросов. Т. к. аргументы для макроса могут быть выражениями, *macro_call* рекурсивна.

Первый параметр процедуры *macro_call* указывает на счётчик ссылок списка лексем, определяющего макрос. Второй параметр содержит любые аргументы, уже разобранные (см. ниже). Третий параметр указывает на символьную лексему, именующую макрос. Если третий параметр есть *null*, макрос определён командой **vardef**, и его имя можно восстановить из префикса и “at” аргументов во втором параметре.

Что за второй параметр? Это просто связанный список однословных элементов, поля́ *info* которых указывают на аргументы. Другими словами, если *arg_list* = *null*, ещё нет считанных аргументов; в противном случае *info*(*arg_list*) указывает на первый считанный аргумент, а *link*(*arg_list*) — на список последующих аргументов (если они есть).

Аргументы типа **expr** суть так называемые капсулы, которые мы обсудим позже, когда мы сосредоточимся на выражениях; их можно легко распознать, потому что их поле *link* равно *void*. Аргументы типа **suffix** и **text** — это списки лексем без счётчиков ссылок.

720. После того, как аргументы считаны, они помещаются в *param_stack*. (Их нельзя поместить в стек раньше, потому что стек растёт и сжимается непредсказуемо в зависимости от того, сколько аргументов получено.) Тогда тело макроса поступает в считыватель; то есть, замещающий текст макроса помещается в вершину входного стека METAFONT'а, чтобы *get_next* далее приступила к его чтению.

```
⟨ Объяви процедуру macro_call 720 ⟩ ≡
⟨ Объяви процедуру print_macro_name 722 ⟩
⟨ Объяви процедуру print_arg 723 ⟩
⟨ Объяви процедуру scan_text_arg 730 ⟩
procedure macro_call(def_ref, arg_list, macro_name : pointer);
        { осуществляет определённую пользователем управляющую последовательность }
  label found;
  var r: pointer;  { текущий узел в списке лексем макросов }
    p, q: pointer;  { для работы со списком }
    n: integer;  { число аргументов }
    l_delim, r_delim: pointer;  { пара разделителей }
    tail: pointer;  { остаток списка аргументов }
  begin r ← link(def_ref); add_mac_ref(def_ref);
  if arg_list = null then n ← 0
  else ⟨ Определи число n уже замещённых аргументов и установи tail на конец списка arg_list 724 ⟩;
  if internal[tracing_macros] > 0 then
    ⟨ Покажи текст раскрываемого макро и существующих аргументов 721 ⟩;
  ⟨ Считай оставшиеся аргументы, если есть; установи r на первую лексему замещающего текста 725 ⟩;
  ⟨ Направь аргументы и замещающий текст в считыватель 736 ⟩;
  end;
```

Этот код используется в разделе 706.

721. ⟨ Покажи текст раскрываемого макро и существующих аргументов 721 ⟩ ≡

```
  begin begin_diagnostic; print_ln; print_macro_name(arg_list, macro_name);
  if n = 3 then print("@#");  { указывает, что макрос с суффиксом }
  show_macro(def_ref, null, 100000);
  if arg_list ≠ null then
    begin n ← 0; p ← arg_list;
    repeat q ← info(p); print_arg(q, n, 0); incr(n); p ← link(p);
    until p = null;
    end;
  end_diagnostic(false);
  end
```

Этот код используется в разделе 720.

722. ⟨ Объяви процедуру *print_macro_name* 722 ⟩ ≡
```
procedure print_macro_name(a, n : pointer);
  var p, q: pointer; { они проходят первую часть a }
  begin if n ≠ null then slow_print(text(n))
  else begin p ← info(a);
    if p = null then slow_print(text(info(info(link(a)))))
    else begin q ← p;
      while link(q) ≠ null do q ← link(q);
      link(q) ← info(link(a)); show_token_list(p, null, 1000, 0); link(q) ← null;
      end;
    end;
  end;
```
Этот код используется в разделе 720.

723. ⟨ Объяви процедуру *print_arg* 723 ⟩ ≡
```
procedure print_arg(q : pointer; n : integer; b : pointer);
  begin if link(q) = void then print_nl("(EXPR")
  else if (b < text_base) ∧ (b ≠ text_macro) then print_nl("(SUFFIX")
    else print_nl("(TEXT");
  print_int(n); print(")<-");
  if link(q) = void then print_exp(q, 1)
  else show_token_list(q, null, 1000, 0);
  end;
```
Этот код используется в разделе 720.

724. ⟨ Определи число n уже замещённых аргументов и установи *tail* на конец списка *arg_list* 724 ⟩ ≡
```
  begin n ← 1; tail ← arg_list;
  while link(tail) ≠ null do
    begin incr(n); tail ← link(tail);
    end;
  end
```
Этот код используется в разделе 720.

725. ⟨ Считай оставшиеся аргументы, если есть; установи r на первую лексему замещающего текста 725 ⟩ ≡
```
  cur_cmd ← comma + 1; { что-нибудь ≠ comma будет сделано }
  while info(r) ⩾ expr_base do
    begin ⟨ Считай разграниченный аргумент, представленный полем info(r) 726 ⟩;
    r ← link(r);
    end;
  if cur_cmd = comma then
    begin print_err("Too␣many␣arguments␣to␣"); print_macro_name(arg_list, macro_name);
    print_char(";"); print_nl("␣␣Missing␣`"); slow_print(text(r_delim));
    print("´␣has␣been␣inserted");
    help3("I´m␣going␣to␣assume␣that␣the␣comma␣I␣just␣read␣was␣a")
    ("right␣delimiter,␣and␣then␣I´ll␣begin␣expanding␣the␣macro.")
    ("You␣might␣want␣to␣delete␣some␣tokens␣before␣continuing."); error;
    end;
  if info(r) ≠ general_macro then ⟨ Считай неразграниченный аргумент (или аргументы) 733 ⟩;
  r ← link(r)
```
Этот код используется в разделе 720.

726. Здесь читателю стоит просмотреть приведённые ранее описания формата списка лексем, уделив особое внимание соглашениям, относящимся только к началу списка лексем макроса.

С другой стороны, читателю придётся принять на веру вопросы разбора выражений в следующей программе. Мы объясним *cur_type* и *cur_exp* позже. (Кое-что в этой программе взаимосвязанно, и неизбежно где-то приходится ходить по кругу.)

```
⟨ Считай разграниченный аргумент, представленный полем info(r) 726 ⟩ ≡
  if cur_cmd ≠ comma then
    begin get_x_next;
    if cur_cmd ≠ left_delimiter then
      begin print_err("Missing␣argument␣to␣");  print_macro_name(arg_list, macro_name);
      help3("That␣macro␣has␣more␣parameters␣than␣you␣thought.")
      ("I´ll␣continue␣by␣pretending␣that␣each␣missing␣argument")
      ("is␣either␣zero␣or␣null.");
      if info(r) ⩾ suffix_base then
        begin cur_exp ← null;  cur_type ← token_list;
        end
      else begin cur_exp ← 0;  cur_type ← known;
        end;
      back_error;  cur_cmd ← right_delimiter;  goto found;
      end;
    l_delim ← cur_sym;  r_delim ← cur_mod;
    end;
  ⟨ Считай аргумент, представленный info(r) 729 ⟩;
  if cur_cmd ≠ comma then ⟨ Проверь, что надлежащий правый разделитель присутствует 727 ⟩;
found: ⟨ Добавь текущее выражение к arg_list 728 ⟩
```

Этот код используется в разделе 725.

727. ⟨ Проверь, что надлежащий правый разделитель присутствует 727 ⟩ ≡

```
  if (cur_cmd ≠ right_delimiter) ∨ (cur_mod ≠ l_delim) then
    if info(link(r)) ⩾ expr_base then
      begin missing_err(",");  help3("I´ve␣finished␣reading␣a␣macro␣argument␣and␣am␣about␣to")
      ("read␣another;␣the␣arguments␣weren´t␣delimited␣correctly.")
      ("You␣might␣want␣to␣delete␣some␣tokens␣before␣continuing.");  back_error;
      cur_cmd ← comma;
      end
    else begin missing_err(text(r_delim));
      help2("I´ve␣gotten␣to␣the␣end␣of␣the␣macro␣parameter␣list.")
      ("You␣might␣want␣to␣delete␣some␣tokens␣before␣continuing.");  back_error;
      end
```

Этот код используется в разделе 726.

728. Параметр **suffix** или **text** будет считан, как список лексем, указанный переменной *cur_exp*, в таком случае мы будем иметь *cur_type* = *token_list*.

```
⟨ Добавь текущее выражение к arg_list 728 ⟩ ≡
  begin p ← get_avail;
  if cur_type = token_list then info(p) ← cur_exp
  else info(p) ← stash_cur_exp;
  if internal[tracing_macros] > 0 then
    begin begin_diagnostic; print_arg(info(p), n, info(r)); end_diagnostic(false);
    end;
  if arg_list = null then arg_list ← p
  else link(tail) ← p;
  tail ← p; incr(n);
  end
```

Этот код используется в разделах 726 и 733.

729.
```
⟨ Считай аргумент, представленный info(r) 729 ⟩ ≡
  if info(r) ⩾ text_base then scan_text_arg(l_delim, r_delim)
  else begin get_x_next;
    if info(r) ⩾ suffix_base then scan_suffix
    else scan_expression;
    end
```

Этот код используется в разделе 726.

730. Параметрами для *scan_text_arg* являются или пара разделителей, или нуль. Последний случай предназначен для текстовых аргументов без разделителей, которые заканчиваются первой точкой с запятой, или **endgroup**, или **end**, и которые не содержатся в группе.

```
⟨ Объяви процедуру scan_text_arg 730 ⟩ ≡
procedure scan_text_arg(l_delim, r_delim : pointer);
  label done;
  var balance: integer;   { превышение l_delim над r_delim }
    p: pointer;   { конец списка }
  begin warning_info ← l_delim; scanner_status ← absorbing; p ← hold_head; balance ← 1;
  link(hold_head) ← null;
  loop begin get_next;
    if l_delim = 0 then
      ⟨ Подправь баланс для неразграниченного аргумента; goto done, если сделано 732 ⟩
    else ⟨ Подправь баланс для разграниченного аргумента; goto done, если сделано 731 ⟩;
    link(p) ← cur_tok; p ← link(p);
    end;
done: cur_exp ← link(hold_head); cur_type ← token_list; scanner_status ← normal;
  end;
```

Этот код используется в разделе 720.

731. ⟨ Подправь баланс для разграниченного аргумента; **goto** *done*, если сделано 731 ⟩ ≡

```
begin if cur_cmd = right_delimiter then
  begin if cur_mod = l_delim then
    begin decr(balance);
    if balance = 0 then goto done;
    end;
  end
else if cur_cmd = left_delimiter then
    if cur_mod = r_delim then incr(balance);
end
```

Этот код используется в разделе 730.

732. ⟨ Подправь баланс для неразграниченного аргумента; **goto** *done*, если сделано 732 ⟩ ≡

```
begin if end_of_statement then  { cur_cmd = semicolon, end_group или stop }
  begin if balance = 1 then goto done
  else if cur_cmd = end_group then decr(balance);
  end
else if cur_cmd = begin_group then incr(balance);
end
```

Этот код используется в разделе 730.

733. ⟨ Считай неразграниченный аргумент (или аргументы) 733 ⟩ ≡

```
begin if info(r) < text_macro then
  begin get_x_next;
  if info(r) ≠ suffix_macro then
    if (cur_cmd = equals) ∨ (cur_cmd = assignment) then get_x_next;
  end;
case info(r) of
primary_macro: scan_primary;
secondary_macro: scan_secondary;
tertiary_macro: scan_tertiary;
expr_macro: scan_expression;
of_macro: ⟨ Считай выражение, за которым следует 'of ⟨primary⟩' 734 ⟩;
suffix_macro: ⟨ Считай суффикс с необязательными разделителями 735 ⟩;
text_macro: scan_text_arg(0, 0);
end;  { нет других случаев }
back_input; ⟨ Добавь текущее выражение к arg_list 728 ⟩;
end
```

Этот код используется в разделе 725.

734. ⟨ Считай выражение, за которым следует '**of** ⟨primary⟩' 734 ⟩ ≡

```
begin scan_expression; p ← get_avail; info(p) ← stash_cur_exp;
if internal[tracing_macros] > 0 then
  begin begin_diagnostic; print_arg(info(p), n, 0); end_diagnostic(false);
  end;
if arg_list = null then arg_list ← p else link(tail) ← p;
tail ← p; incr(n);
if cur_cmd ≠ of_token then
  begin missing_err("of"); print("␣for␣"); print_macro_name(arg_list, macro_name);
  help1("I´ve␣got␣the␣first␣argument;␣will␣look␣now␣for␣the␣other.");
  back_error;
  end;
get_x_next; scan_primary;
end
```

Этот код используется в разделе 733.

735. ⟨ Считай суффикс с необязательными разделителями 735 ⟩ ≡

```
begin if cur_cmd ≠ left_delimiter then l_delim ← null
else begin l_delim ← cur_sym; r_delim ← cur_mod; get_x_next;
  end;
scan_suffix;
if l_delim ≠ null then
  begin if (cur_cmd ≠ right_delimiter) ∨ (cur_mod ≠ l_delim) then
    begin missing_err(text(r_delim));
    help2("I´ve␣gotten␣to␣the␣end␣of␣the␣macro␣parameter␣list.")
    ("You␣might␣want␣to␣delete␣some␣tokens␣before␣continuing.");
    back_error;
    end;
  get_x_next;
  end;
end
```

Этот код используется в разделе 733.

736. Перед тем, как поместить новый список лексем во входной стек, желательно очистить все исчерпавшиеся списки лексем. Тогда макрос пользователя, заканчивающийся вызывом самого себя, не будет требовать в стеке неограниченного места.

⟨ Направь аргументы и замещающий текст в считыватель 736 ⟩ ≡

```
while token_state ∧ (loc = null) do end_token_list;  { сохраняет место в стеке }
if param_ptr + n > max_param_stack then
  begin max_param_stack ← param_ptr + n;
  if max_param_stack > param_size then overflow("parameter␣stack␣size", param_size);
  end;
begin_token_list(def_ref, macro); name ← macro_name; loc ← r;
if n > 0 then
  begin p ← arg_list;
  repeat param_stack[param_ptr] ← info(p); incr(param_ptr); p ← link(p);
  until p = null;
  flush_list(arg_list);
  end
```

Этот код используется в разделе 720.

737. Иногда нужно поместить один аргумен в *param_stack*. Это делает подпрограмма *stack_argument*.

```
procedure stack_argument(p : pointer);
  begin if param_ptr = max_param_stack then
    begin incr(max_param_stack);
    if max_param_stack > param_size then overflow("parameter␣stack␣size", param_size);
    end;
  param_stack[param_ptr] ← p; incr(param_ptr);
  end;
```

738. Обработка условий. [Conditional processing] Давайте, рассмотрим обработку команд **if**.

Условия могут размещаться внутри других условий, для этого есть стек, независимый от других стеков. Четыре глобальных переменных представляют вершину стека условий: *cond_ptr* указывает на помещённые в стек элементы, если есть, *cur_if* указывает, обрабатывается ли **if** или **elseif**, *if_limit* указывает наибольший синтаксически допустимый код команды *fi_or_else*, а *if_line* хранит номер строки, в которой текущее условие началось.

Если сейчас не обрабатываются условия, стек условий имеет особое состояние $cond\_ptr = null$, $if\_limit = normal$, $cur\_if = 0$, $if\_line = 0$. Иначе *cond_ptr* указывает на двусловный узел; поля *type*, *name_type* и *link* первого слова содержат *if_limit*, *cur_if* и *cond_ptr* на следующем уровне, а второе слово содержит соответствующее значение *if_line*.

define *if_node_size* = 2 { число слов в элементе стека для условий }
define *if_line_field*(#) ≡ *mem*[# + 1].*int*
define *if_code* = 1 { код для оцениваемого **if** }
define *fi_code* = 2 { код для **fi** }
define *else_code* = 3 { код для **else** }
define *else_if_code* = 4 { код для **elseif** }

⟨ Общие переменные 13 ⟩ +≡
cond_ptr: *pointer*; { вершина стека условий }
if_limit: *normal* .. *else_if_code*; { верхняя граница кодов *fi_or_else* }
cur_if: *small_number*; { вид обрабатываемого условия }
if_line: *integer*; { строка, где условие началось }

739. ⟨ Установи начальные значения ключевых переменных 21 ⟩ +≡
cond_ptr ← *null*; *if_limit* ← *normal*; *cur_if* ← 0; *if_line* ← 0;

740. ⟨ Помести каждый примитив METAFONT'а в хеш-таблицу 192 ⟩ +≡
primitive("if", *if_test*, *if_code*);
primitive("fi", *fi_or_else*, *fi_code*); *eqtb*[*frozen_fi*] ← *eqtb*[*cur_sym*];
primitive("else", *fi_or_else*, *else_code*);
primitive("elseif", *fi_or_else*, *else_if_code*);

741. ⟨ Случаи *print_cmd_mod* для символьной печати примитивов 212 ⟩ +≡
if_test, *fi_or_else*: **case** *m* **of**
if_code: *print*("if");
fi_code: *print*("fi");
else_code: *print*("else");
othercases *print*("elseif")
endcases;

742. Вот процедура, пропускающая текст до **elseif**, **else** или **fi** на нулевом уровне вложения конструкций **if** ... **fi**. По окончании её работы *cur_mod* будет указывать на найденную лексему.

Коды команд *if_test* и *fi_or_else* — наименьшие у METAFONT'а; это немного облегчает задачу пропуска.

```
procedure pass_text;
  label done;
  var l: integer;
  begin scanner_status ← skipping; l ← 0; warning_info ← line;
  loop begin get_next;
    if cur_cmd ⩽ fi_or_else then
      if cur_cmd < fi_or_else then incr(l)
      else begin if l = 0 then goto done;
        if cur_mod = fi_code then decr(l);
        end
    else ⟨ Уменьши счётчик ссылок строк, если текущая лексема суть строка 743 ⟩;
    end;
done: scanner_status ← normal;
  end;
```

743. ⟨ Уменьши счётчик ссылок строк, если текущая лексема суть строка 743 ⟩ ≡

```
  if cur_cmd = string_token then delete_str_ref(cur_mod)
```

Этот код используется в разделах 83, 742, 991 и 1016.

744. Когда мы начнём обрабатывать новое **if**, мы установим *if_limit* ← *if_code*; затем, если встретится **elseif**, **else** или **fi** до того, как текущее условие **if** будет вычислено, будет вставлено двоеточие. Иначе бы конструкции вроде '`if fi`' приведили METAFONT в замешательство.

⟨ Помести в стек условий 744 ⟩ ≡

```
  begin p ← get_node(if_node_size); link(p) ← cond_ptr; type(p) ← if_limit; name_type(p) ← cur_if;
  if_line_field(p) ← if_line; cond_ptr ← p; if_limit ← if_code; if_line ← line; cur_if ← if_code;
  end
```

Этот код используется в разделе 748.

745. ⟨ Извлеки из стека условий 745 ⟩ ≡

```
  begin p ← cond_ptr; if_line ← if_line_field(p); cur_if ← name_type(p); if_limit ← type(p);
  cond_ptr ← link(p); free_node(p, if_node_size);
  end
```

Этот код используется в разделах 748, 749 и 751.

746. Вот процедура, изменяющая код *if_limit* в соответствии с данным значением *cond_ptr*.

```
procedure change_if_limit(l : small_number; p : pointer);
  label exit;
  var q: pointer;
  begin if p = cond_ptr then if_limit ← l  { это лёгкий случай }
  else begin q ← cond_ptr;
    loop begin if q = null then confusion("if");
      if link(q) = p then
        begin type(q) ← l; return;
        end;
      q ← link(q);
      end;
    end;
exit: end;
```

747. Пользователь должен поместить двоеточие в соответствующие части условных операторов. Поэтому, METAFONT должен проверить их наличие.

```
procedure check_colon;
  begin if cur_cmd ≠ colon then
    begin missing_err(":");
    help2("There␣should´ve␣been␣a␣colon␣after␣the␣condition.")
    ("I␣shall␣pretend␣that␣one␣was␣there."); back_error;
    end;
  end;
```

748. Условие начинает обрабатываться, когда процедура *get_x_next* встречает команду *if_test*; в этом случае *get_x_next* вызывает рекурсивную процедуру *conditional*.

```
procedure conditional;
  label exit, done, reswitch, found;
  var save_cond_ptr: pointer;  { cond_ptr соответсвует этому условию }
    new_if_limit: fi_code .. else_if_code;  { будущее значение переменной if_limit }
    p: pointer;  { временный регистр }
  begin ⟨ Помести в стек условий 744 ⟩; save_cond_ptr ← cond_ptr;
reswitch: get_boolean; new_if_limit ← else_if_code;
  if internal[tracing_commands] > unity then ⟨ Покажи булево значение cur_exp 750 ⟩;
found: check_colon;
  if cur_exp = true_code then
    begin change_if_limit(new_if_limit, save_cond_ptr); return;  { ожидать elseif, else или fi }
    end;
  ⟨ Пропусти до elseif, или else, или fi, затем goto done 749 ⟩;
done: cur_if ← cur_mod; if_line ← line;
  if cur_mod = fi_code then ⟨ Извлеки из стека условий 745 ⟩
  else if cur_mod = else_if_code then goto reswitch
    else begin cur_exp ← true_code; new_if_limit ← fi_code; get_x_next; goto found;
      end;
exit: end;
```

749. В выражениях вроде '**if if true**: 0 = 1: *foo* **else**: *bar* **fi**', первое **else**, которое мы встретим, узнав, что **if** ложно, не то **else**, которое мы ищем. Поэтому, нужна следующая любопытная логика.

```
⟨ Пропусти до elseif, или else, или fi, затем goto done 749 ⟩ ≡
  loop begin pass_text;
    if cond_ptr = save_cond_ptr then goto done
    else if cur_mod = fi_code then ⟨ Извлеки из стека условий 745 ⟩;
    end
```

Этот код используется в разделе 748.

750.

```
⟨ Покажи булево значение cur_exp 750 ⟩ ≡
  begin begin_diagnostic;
  if cur_exp = true_code then print("{true}") else print("{false}");
  end_diagnostic(false);
  end
```

Этот код используется в разделе 748.

751. Обработка условий завершена, за исключением следующего кода, который, в сущности, является частью *get_x_next*. Он начинает действовать, когда обнаруживаются **elseif**, **else** или **fi**.

```
⟨ Заверши текущее условие и пропусти до fi 751 ⟩ ≡
  if cur_mod > if_limit then
    if if_limit = if_code then { ещё не оцененное условие }
      begin missing_err(":"); back_input; cur_sym ← frozen_colon; ins_error;
      end
    else begin print_err("Extra␣"); print_cmd_mod(fi_or_else, cur_mod);
      help1("I´m␣ignoring␣this;␣it␣doesn´t␣match␣any␣if."); error;
      end
  else begin while cur_mod ≠ fi_code do pass_text; { перейти к fi }
    ⟨ Извлеки из стека условий 745 ⟩;
    end
```

Этот код используется в разделе 707.

752. Повторения. [Iterations] Чтобы завершить разбор *get_x_next*, нам нужно рассмотреть, что METAFONT делает, когда видит **for**, **forsuffixes** или **forever**.

Глобальная переменная *loop_ptr* отслеживает обрабатываемые сейчас циклы **for**. При $loop\_ptr = null$, ни один цикл не обрабатывается; иначе $info(loop\_ptr)$ указывает на повторяющийся текст текущего (самого внутреннего) цикла, а $link(loop\_ptr)$ указывает на данные любых внешних циклов, объемлющих текущий.

Узел управления циклом [loop-control node] имеет также два других поля: *loop_type* и *loop_list*, содержимое которорых зависит от вида цикла:

$loop\_type(loop\_ptr) = null$ означает, что $loop\_list(loop\_ptr)$ указывает на список однословных узлов, поля́ *info* которых указывают на оставшиеся значения аргументов списка суффиксов и списка выражений.

$loop\_type(loop\_ptr) = void$ значит, что текущий цикл — '**forever**'.

$loop\_type(loop\_ptr) = p > void$ значит, что $value(p)$, $step\_size(p)$ и $final\_value(p)$ содержат данные арифметической прогрессии.

В последнем случае, p указывает на «узел прогрессии» [“progression node”], первое слово которого не используется. (Ни какое значение нельзя здесь хранить, т. к. поле ссылки слов в области динамической памяти не может быть произвольным.)

```
define loop_list_loc(#) ≡ # + 1   { где находится поле loop_list }
define loop_type(#) ≡ info(loop_list_loc(#))   { вид цикла for }
define loop_list(#) ≡ link(loop_list_loc(#))   { остающиеся элементы списка }
define loop_node_size = 2   { число слов в узле управления циклом }
define progression_node_size = 4   { число слов в узле прогрессии }
define step_size(#) ≡ mem[# + 2].sc   { разность арифметической прогрессии }
define final_value(#) ≡ mem[# + 3].sc   { конечное значение арифметической прогрессии }
```

⟨ Общие переменные 13 ⟩ +≡
```
loop_ptr: pointer;   { вершина стека узлов управления циклом }
```

753. ⟨ Установи начальные значения ключевых переменных 21 ⟩ +≡
```
loop_ptr ← null;
```

754. Если выражения, определяющие арифметическую прогрессию в цикле **for**, не имеют известных числовых значений, подпрограмма *bad_for* сообщает об этом пользователю.

```
procedure bad_for(s : str_number);
  begin disp_err(null, "Improper␣");   { покажи плохое выражение над сообщением }
  print(s); print("␣has␣been␣replaced␣by␣0"); help4("When␣you␣say␣`for␣x=a␣step␣b␣until␣c´,")
  ("the␣initial␣value␣`a´␣and␣the␣step␣size␣`b´")
  ("and␣the␣final␣value␣`c´␣must␣have␣known␣numeric␣values.")
  ("I´m␣zeroing␣this␣one.␣Proceed,␣with␣fingers␣crossed."); put_get_flush_error(0);
  end;
```

755. Вот, что делает METAFONTпосле считывания **for**, **forsuffixes** или **forever**. (Этот код требует некоторого знакомства с подпрограммами разбора выражений, которые мы ещё не обсудили, но он, кажется, принадлежит данной части программы, даже если бы автор написал его позже. Читатель может пожелать вернуться к нему.)

```
procedure begin_iteration;
  label continue, done, found;
  var m: halfword;  { expr_base (for) или suffix_base (forsuffixes) }
    n: halfword;  { хеш-адрес текущего символа }
    p, q, s, pp: pointer;  { регистры для работы со ссылками }
  begin m ← cur_mod; n ← cur_sym; s ← get_node(loop_node_size);
  if m = start_forever then
    begin loop_type(s) ← void; p ← null; get_x_next; goto found;
    end;
  get_symbol; p ← get_node(token_node_size); info(p) ← cur_sym; value(p) ← m;
  get_x_next;
  if (cur_cmd ≠ equals) ∧ (cur_cmd ≠ assignment) then
    begin missing_err("=");
    help3("The␣next␣thing␣in␣this␣loop␣should␣have␣been␣`=´␣or␣`:=´.")
    ("But␣don´t␣worry;␣I´ll␣pretend␣that␣an␣equals␣sign")
    ("was␣present,␣and␣I´ll␣look␣for␣the␣values␣next.");
    back_error;
    end;
  ⟨ Считай значения для использования в цикле 764 ⟩;
found: ⟨ Проверь наличие двоеточия 756 ⟩;
  ⟨ Считай текст цикла и помести его в стек управления циклами 758 ⟩;
  resume_iteration;
  end;
```

756. ⟨ Проверь наличие двоеточия 756 ⟩ ≡

```
  if cur_cmd ≠ colon then
    begin missing_err(":");
    help3("The␣next␣thing␣in␣this␣loop␣should␣have␣been␣a␣`:´.")
    ("So␣I´ll␣pretend␣that␣a␣colon␣was␣present;")
    ("everything␣from␣here␣to␣`endfor´␣will␣be␣iterated."); back_error;
    end
```

Этот код используется в разделе 755.

757. Мы добавляем особую лексему *frozen_repeat_loop* на месте команды '**endfor**' в конце цикла. Она поступит в считыватель METAFONT'а, когда нужно будет повторить цикл.

(Если пользователь попытается сделать что-нибудь вроде '**for** ... **let endfor**', ему помешает подпрограмма *get_symbol*, которая оставит замороженные лексемы нетронутыми. Более того, сама подпрограмма *frozen_repeat_loop* является **outer** лексемой, поэтому она не может быть уничтожена случайно.)

758. ⟨ Считай текст цикла и помести его в стек управления циклами 758 ⟩ ≡

```
  q ← get_avail; info(q) ← frozen_repeat_loop; scanner_status ← loop_defining; warning_info ← n;
  info(s) ← scan_toks(iteration, p, q, 0); scanner_status ← normal;
  link(s) ← loop_ptr; loop_ptr ← s
```

Этот код используется в разделе 755.

759. ⟨ Настрой таблицы (делает только INIMF) 176 ⟩ +≡

```
  eq_type(frozen_repeat_loop) ← repeat_loop + outer_tag; text(frozen_repeat_loop) ← "␣ENDFOR";
```

760. Текст цикла вставляется в считывающий аппарат METAFONT'а подпрограммой *resume_iteration*.

```
procedure resume_iteration;
  label not_found, exit;
  var p, q: pointer;  { регистры ссылок }
  begin p ← loop_type(loop_ptr);
  if p > void then  { p указывает на узел прогрессии }
    begin cur_exp ← value(p);
    if ⟨ Арифметическая прогрессия закончилась 761 ⟩ then goto not_found;
    cur_type ← known; q ← stash_cur_exp;  { делает q аргументом expr }
    value(p) ← cur_exp + step_size(p);  { устанавливает value(p) на следующее повторение }
    end
  else if p < void then
      begin p ← loop_list(loop_ptr);
      if p = null then goto not_found;
      loop_list(loop_ptr) ← link(p); q ← info(p); free_avail(p);
      end
    else begin begin_token_list(info(loop_ptr), forever_text); return;
      end;
  begin_token_list(info(loop_ptr), loop_text); stack_argument(q);
  if internal[tracing_commands] > unity then ⟨ Отслеживай начало цикла 762 ⟩;
  return;
not_found: stop_iteration;
exit: end;
```

761. ⟨ Арифметическая прогрессия закончилась 761 ⟩ ≡

```
((step_size(p) > 0) ∧ (cur_exp > final_value(p))) ∨ ((step_size(p) < 0) ∧ (cur_exp < final_value(p)))
```

Этот код используется в разделе 760.

762. ⟨ Отслеживай начало цикла 762 ⟩ ≡

```
begin begin_diagnostic; print_nl("{loop␣value=");
if (q ≠ null) ∧ (link(q) = void) then print_exp(q, 1)
else show_token_list(q, null, 50, 0);
print_char("}"); end_diagnostic(false);
end
```

Этот код используется в разделе 760.

763. Уровень управления циклом исчезает, когда *resume_iteration* решит не повторять [цикл], или когда [команда] **exitif** удалит текст цикла из входного стека.

```
procedure stop_iteration;
  var p,q: pointer;  { как обычно }
  begin p ← loop_type(loop_ptr);
  if p > void then free_node(p, progression_node_size)
  else if p < void then
      begin q ← loop_list(loop_ptr);
      while q ≠ null do
        begin p ← info(q);
        if p ≠ null then
          if link(p) = void then  { это параметр expr }
            begin recycle_value(p); free_node(p, value_node_size);
            end
          else flush_token_list(p);  { это параметр suffix или text }
        p ← q; q ← link(q); free_avail(p);
        end;
      end;
  p ← loop_ptr; loop_ptr ← link(p); flush_token_list(info(p)); free_node(p, loop_node_size);
  end;
```

764. Теперь, когда мы знаем всё об управлении циклами, мы можем завершить оставшуюся часть *begin_iteration*, что мы и сделаем.

Следующий код выполняется после того, как прочитан символ '=' в конструкции **for** (если $m = expr\_base$) или конструкции **forsuffixes** (если $m = suffix\_base$).

```
⟨ Считай значения для использования в цикле 764 ⟩ ≡
  loop_type(s) ← null; q ← loop_list_loc(s); link(q) ← null;  { link(q) = loop_list(s) }
  repeat get_x_next;
    if m ≠ expr_base then scan_suffix
    else begin if cur_cmd ⩾ colon then
        if cur_cmd ⩽ comma then goto continue;
      scan_expression;
      if cur_cmd = step_token then
        if q = loop_list_loc(s) then ⟨ Готовься к конструкции step-until и goto done 765 ⟩;
      cur_exp ← stash_cur_exp;
      end;
    link(q) ← get_avail; q ← link(q); info(q) ← cur_exp; cur_type ← vacuous;
  continue: until cur_cmd ≠ comma;
done:
```

Этот код используется в разделе 755.

765. ⟨ Готовься к конструкции step-until и **goto** *done* 765 ⟩ ≡

```
begin if cur_type ≠ known then bad_for("initial␣value");
pp ← get_node(progression_node_size); value(pp) ← cur_exp;
get_x_next; scan_expression;
if cur_type ≠ known then bad_for("step␣size");
step_size(pp) ← cur_exp;
if cur_cmd ≠ until_token then
  begin missing_err("until");
  help2("I␣assume␣you␣meant␣to␣say␣`until´␣after␣`step´.")
  ("So␣I´ll␣look␣for␣the␣final␣value␣and␣colon␣next."); back_error;
  end;
get_x_next; scan_expression;
if cur_type ≠ known then bad_for("final␣value");
final_value(pp) ← cur_exp; loop_type(s) ← pp; goto done;
end
```

Этот код используется в разделе 764.

⟨ Арифметическая прогрессия закончилась 761 ⟩ Используется в разделе 760.
⟨ Безучастные *independent* случаи в капсуле p 926 ⟩ Используется в разделе 922.
⟨ Безучастные *independent* случаи в текущем выражении 927 ⟩ Используется в разделе 922.
⟨ Введи из внешнего файла; **goto** *restart*, если вводить нечего, или **return**, если несимвольная лексема найдена 669 ⟩ Используется в разделе 667.
⟨ Введи из списка лексем; **goto** *restart*, если конец списка, или если параметр нужно раскрыть, или **return**, если несимвольная лексема найдена 676 ⟩ Используется в разделе 667.
⟨ Верни внешнюю символьную лексему, чтобы её можно было повторно прочитать 662 ⟩ Используется в разделе 661.
⟨ Верни подходящий ответ на основе z и *octant* 141 ⟩ Используется в разделе 139.
⟨ Верни список *sorted*(p) в сортировку 345 ⟩ Используется в разделе 344.
⟨ Включи содержимое строки *pp* в строку p 368 ⟩ Используется в разделе 366.
⟨ Включи список *temp_head* в *sorted*(h) 347 ⟩ Используется в разделе 346.
⟨ Внеси в файл протокола размеры подфайлов TFM файла 1141 ⟩ Используется в разделе 1134.
⟨ Внеси член из p, добавь f раз соответствующий член из q 595 ⟩ Используется в разделе 594.
⟨ Внеси член из p, добавь соответствующий член из q 598 ⟩ Используется в разделе 597.
⟨ Внеси член из q, умноженный на f 596 ⟩ Используется в разделе 594.
⟨ Вопи о делении на нуль 950 ⟩ Используется в разделе 948.
⟨ Вставь вес границы для границы m, если вес новой точки изменился 350 ⟩ Используется в разделе 349.
⟨ Вставь восходящие границы для линии 375 ⟩ Используется в разделе 374.
⟨ Вставь горизонтальные границы веса w между m и mm 362 ⟩ Используется в разделе 358.
⟨ Вставь горизонтальные границы, определённые смежными строками p, q, и уничтожь строку p 358 ⟩ Используется в разделе 354.
⟨ Вставь границы окатнта и вычисли округлённое число 450 ⟩ Используется в разделе 402.
⟨ Вставь дополнительные узлы границ, затем **goto** *done* 458 ⟩ Используется в разделе 452.
⟨ Вставь линейный двойной сегмент, чтобы подойти к верному смещению 521 ⟩ Используется в разделе 518.
⟨ Вставь линейный сегмент, чтобы подойти к верному смещению 515 ⟩ Используется в разделе 512.
⟨ Вставь нисходящие границы для линии 376 ⟩ Используется в разделе 374.
⟨ Вставь новую символьную лексему после p, затем установи p на неё и **goto** *found* 207 ⟩ Используется в разделе 205.
⟨ Вставь новую строку для направления (u, v) между p и q 535 ⟩ Используется в разделе 531.
⟨ Вставь новые данные с терминала и **return** 82 ⟩ Используется в разделе 79.
⟨ Вставь новые двойные перемещения огибающей в данных точек 523 ⟩ Используется в разделе 518.
⟨ Вставь новые перемещения огибающей в данных точек 517 ⟩ Используется в разделе 512.
⟨ Вставь один или больше узлов границ октанта сразу перед q 452 ⟩ Используется в разделе 450.
⟨ Вставь пустые строки сверху и снизу и установи p на новую верхнюю строку 355 ⟩ Используется в разделе 354.
⟨ Вставь ровно *n_min*$($*cur_edges*$) - nl$ пустых строк сверху 331 ⟩ Используется в разделе 329.
⟨ Вставь ровно *n_min*$($*cur_edges*$) - nl$ пустых строк снизу 330 ⟩ Используется в разделе 329.
⟨ Вставь суффиксный или текстовый параметр и **goto** *restart* 677 ⟩ Используется в разделе 676.
⟨ Вставь узел дроби, разбивая кубический сплайн 986 ⟩ Используется в разделе 985.
⟨ Выбери зависимую переменную для замещения исчезающей независимой переменной, и измени соответственно все оставшиеся зависимости 815 ⟩ Используется в разделе 812.
⟨ Выбери управляющие точки для пути и помести итог в *cur_exp* 891 ⟩ Используется в разделе 869.
⟨ Выведи байты данных символа, далее выведи сами размеры 1136 ⟩ Используется в разделе 1134.
⟨ Выведи кусочки расширяемых символов и параметры метрик шрифта 1140 ⟩ Используется в разделе 1134.
⟨ Выведи ответ, v (который мог стать *known*) 934 ⟩ Используется в разделе 932.
⟨ Выведи программу лигатур и кернов 1139 ⟩ Используется в разделе 1134.
⟨ Выведи размеры подфайлов и байты заголовка 1135 ⟩ Используется в разделе 1134.
⟨ Выведи символ, представленный в *cur_edges* 1167 ⟩ Используется в разделе 1165.

⟨ Выведи статистику об этой работе 1208 ⟩ Используется в разделе 1205.
⟨ Выведи точки края строки p к краю шрифта n 1169 ⟩ Используется в разделе 1167.
⟨ Выгрузи динамическую память 1194 ⟩ Используется в разделе 1186.
⟨ Выгрузи константы для проверки целостности 1190 ⟩ Используется в разделе 1186.
⟨ Выгрузи несколько вещей и завершающее контрольное слово 1198 ⟩ Используется в разделе 1186.
⟨ Выгрузи пул строк 1192 ⟩ Используется в разделе 1186.
⟨ Выгрузи таблицу эквивалентов и хеш-таблицу 1196 ⟩ Используется в разделе 1186.
⟨ Выдели весь узел p и **goto** *found* 171 ⟩ Используется в разделе 169.
⟨ Выдели из вершины узла p и **goto** *found* 170 ⟩ Используется в разделе 169.
⟨ Выйди из цикла, если пришло время 713 ⟩ Используется в разделе 707.
⟨ Выйди к *found*, если восточное направление встретилось в узловой точке p 544 ⟩
Используется в разделе 541.
⟨ Выйди к *found*, если кривая, производные которой указаны $x1, x2, x3, y1, y2, y3$, едёт на восток в некоторое время tt 546 ⟩ Используется в разделе 541.
⟨ Выйди к *found*, если производная $B(x_1, x_2, x_3; t)$ стала $\geqslant 0$ 549 ⟩ Используется в разделе 548.
⟨ Выйди раньше времени из итерации 714 ⟩ Используется в разделе 713.
⟨ Выполни код c и **return**, если сделано 79 ⟩ Используется в разделе 78.
⟨ Выполни команду *skip_to* и **goto** *done* 1110 ⟩ Используется в разделе 1107.
⟨ Вычисли $f = \lfloor 2^{16}(1 + p/q) + \frac{1}{2} \rfloor$ 115 ⟩ Используется в разделе 114.
⟨ Вычисли $f = \lfloor 2^{28}(1 + p/q) + \frac{1}{2} \rfloor$ 108 ⟩ Используется в разделе 107.
⟨ Вычисли $p = \lfloor qf/2^{16} + \frac{1}{2} \rfloor - q$ 113 ⟩ Используется в разделе 112.
⟨ Вычисли $p = \lfloor qf/2^{28} + \frac{1}{2} \rfloor - q$ 111 ⟩ Используется в разделе 109.
⟨ Вычисли волшебные значения смещений 365 ⟩ Используется в разделе 354.
⟨ Вычисли входящее и выходящее направления 457 ⟩ Используется в разделе 454.
⟨ Вычисли данное значение величины θ_n и **goto** *found* 292 ⟩ Используется в разделе 284.
⟨ Вычисли значения $aa = A_k/B_k$, $bb = D_k/C_k$, $dd = (3 - \alpha_{k-1})d_{k,k+1}$, $ee = (3 - \beta_{k+1})d_{k-1,k}$ и $cc = (B_k - u_{k-1}A_k)/B_k$ 288 ⟩ Используется в разделе 287.
⟨ Вычисли значения v_k и w_k 290 ⟩ Используется в разделе 287.
⟨ Вычисли значения x до и после на основе текущего пера 435 ⟩ Используется в разделе 434.
⟨ Вычисли значения y до и после на основе текущего пера 438 ⟩ Используется в разделе 437.
⟨ Вычисли код октанта; наклони и поверни координаты (x, y) 489 ⟩ Используется в разделе 488.
⟨ Вычисли компромиссное *pen_edge* 443 ⟩ Используется в разделе 442.
⟨ Вычисли контрольную сумму в $(b1, b2, b3, b4)$ 1132 ⟩ Используется в разделе 1131.
⟨ Вычисли отношение $ff = C_k/(C_k + B_k - u_{k-1}A_k)$ 289 ⟩ Используется в разделе 287.
⟨ Вычисли половину эллипса, отражая уже вычисленную четверть 536 ⟩ Используется в разделе 527.
⟨ Вычисли пробные коэффициенты $(t0, t1, t2)$ для $s(t)$ по отношению к s_k или s_{k-1} 498 ⟩
Используется в разделах 497 и 503.
⟨ Вычисли расстояние d от класса 0 до края эллипса в направлении (u, v), времена $\sqrt{u^2 + v^2}$, округлены до ближайшего целого 533 ⟩ Используется в разделе 531.
⟨ Вычисли смещение программы лигатур/кернов и внедри левую метку границы 1137 ⟩
Используется в разделе 1135.
⟨ Вычисли смещения между экранными и действительными координатами 576 ⟩
Используется в разделе 574.
⟨ Вычисли углы поворотов ψ_k и расстояния $d_{k,k+1}$; установи n на длину пути 281 ⟩
Используется в разделе 278.
⟨ Вычисли хеш-код h 208 ⟩ Используется в разделе 205.
⟨ Вычисли хорошую координату на диагональном перемещении 442 ⟩ Используется в разделе 441.
⟨ Вычисли целые α, β, γ для координат вершин 530 ⟩ Используется в разделе 528.
⟨ Вычти угол z из (x, y) 147 ⟩ Используется в разделе 145.
⟨ Готовься вычислить производную; **goto** *not_found*, если текущий сплайн мёртвый 496 ⟩
Используется в разделе 494.
⟨ Готовься заполнить контур и заполни его 1062 ⟩ Используется в разделе 1059.

⟨ Готовься к замыканию циклического пути 886 ⟩ Используется в разделе 869.
⟨ Готовься к конструкции step-until и **goto** *done* 765 ⟩ Используется в разделе 764.
⟨ Дай совет пользователю и **return** 78 ⟩ Используется в разделе 77.
⟨ Дай сообщения об ошибках, если *bad_char* или $n \geqslant 4096$ 914 ⟩ Используется в разделе 913.
⟨ Делай всё необходимое, когда y — постоянная; **return** или **goto** *continue*, если мёртвый сплайн от p до q удалён 417 ⟩ Используется в разделе 413.
⟨ Делай название 994 ⟩ Используется в разделе 993.
⟨ Делай оператор, который не начинается с выражения 992 ⟩ Используется в разделе 989.
⟨ Делай перемещения для текущего октанта 468 ⟩ Используется в разделе 465.
⟨ Делай перемещения для текущего подинтервала; если бисекция нужна, помести второй подинтервал в стек, и **goto** *continue*, чтобы обработать первый подинтервал 314 ⟩ Используется в разделе 311.
⟨ Делай различные уравнения и **goto** *done* 1005 ⟩ Используется в разделе 1003.
⟨ Делай уравнение, присвоение, название или ‘⟨ выражение ⟩ **endgroup**’ 993 ⟩ Используется в разделе 989.
⟨ Директивы компилятора 9 ⟩ Используется в разделе 4.
⟨ Для каждого из восьми случаев измени соответствующие поля в *cur_exp* и **goto** *done*; но ни чего не делай, если капсула p не подходящего типа 957 ⟩ Используется в разделе 955.
⟨ Для каждого типа t сделай уравнение и **goto** *done*, если *cur_type* совместимо с t 1003 ⟩ Используется в разделе 1001.
⟨ Добавь вклад узла q к общему весу, и установи $q \leftarrow link(q)$ 370 ⟩ Используется в разделах 369 и 369.
⟨ Добавь границы для второго или третьего октантов, затем **goto** *done* 383 ⟩ Используется в разделе 378.
⟨ Добавь границы для первого или четвёртого октантов, затем **goto** *done* 381 ⟩ Используется в разделе 378.
⟨ Добавь границы для пятого или восьмого октантов, затем **goto** *done* 382 ⟩ Используется в разделе 378.
⟨ Добавь границы для шестого или седьмого октантов, затем **goto** *done* 384 ⟩ Используется в разделе 378.
⟨ Добавь известное $value(p)$ к члену постоянной v 933 ⟩ Используется в разделе 932.
⟨ Добавь известное значение к члену постоянной $dep\_list(p)$ 931 ⟩ Используется в разделе 930.
⟨ Добавь операнд p в список зависимостей v 932 ⟩ Используется в разделе 930.
⟨ Добавь правый операнд в список p 1009 ⟩ Используется в разделе 1006.
⟨ Добавь список зависимостей pp типа tt к списку зависимостей p типа t 1010 ⟩ Используется в разделе 1009.
⟨ Добавь текущее выражение к *arg_list* 728 ⟩ Используется в разделах 726 и 733.
⟨ Дополни x координаты сплайна между p и q 409 ⟩ Используется в разделе 407.
⟨ Дополни y координаты сплайна между pp и qq 414 ⟩ Используется в разделах 413 и 417.
⟨ Дополни некоторые края другими 1061 ⟩ Используется в разделе 1059.
⟨ Дополнительные случаи двуместных операторов 936, 940, 941, 948, 951, 952, 975, 983, 988 ⟩ Используется в разделе 922.
⟨ Дополнительные случаи для унарных операторов 905, 906, 907, 909, 912, 915, 917, 918, 920, 921 ⟩ Используется в разделе 898.
⟨ Другие локальные переменные для *disp_edges* 580 ⟩ Используется в разделе 577.
⟨ Другие локальные переменные для *fill_envelope* 511 ⟩ Используется в разделах 506 и 518.
⟨ Другие локальные переменные для *find_direction_time* 542 ⟩ Используется в разделе 539.
⟨ Другие локальные переменные для *make_choices* 280 ⟩ Используется в разделе 269.
⟨ Другие локальные переменные для *make_spec* 453 ⟩ Используется в разделе 402.
⟨ Другие локальные переменные для *offset_prep* 495 ⟩ Используется в разделе 491.
⟨ Другие локальные переменные для *scan_primary* 831, 836, 843 ⟩ Используется в разделе 823.
⟨ Другие локальные переменные для *solve_choices* 286 ⟩ Используется в разделе 284.
⟨ Другие локальные переменные для *xy_swap_edges* 357, 363 ⟩ Используется в разделе 354.
⟨ Если последовательные узловые точки равны, объедини их явно 271 ⟩ Используется в разделе 269.
⟨ Если текущее преобразование полностью известно, оставь его в глобальных переменных; иначе **return** 956 ⟩ Используется в разделе 953.

⟨ Если узел q — точка перехода для координат x, вычисли и сохрани её координаты до-и-после 434 ⟩
Используется в разделе 433.
⟨ Если узел q — точка перехода для координат y, вычисли и сохрани её координаты до-и-после 437 ⟩
Используется в разделе 433.
⟨ Если узел q — точка перехода между октантами, вычисли и сохрани её координаты до-и-после 441 ⟩
Используется в разделе 440.
⟨ Заверши GF файл 1182 ⟩ Используется в разделе 1206.
⟨ Заверши TFM и GF файлы 1206 ⟩ Используется в разделе 1205.
⟨ Заверши TFM файл 1134 ⟩ Используется в разделе 1206.
⟨ Заверши операцию заполнения контура 1064 ⟩ Используется в разделе 1062.
⟨ Заверши получение символьной лексемы в *cur_sym*; **goto** *restart*, если она недопустима 668 ⟩
Используется в разделе 667.
⟨ Заверши процесс разделения смещений 503 ⟩ Используется в разделе 494.
⟨ Заверши связывание узлов смещений, и повтори граничные узлы смещений, если нужно 483 ⟩
Используется в разделе 481.
⟨ Заверши сообщение об ошибке, и установи *cur_sym* на лексему, которая может помочь оправиться от ошибки 664 ⟩ Используется в разделе 663.
⟨ Заверши текущее условие и пропусти до **fi** 751 ⟩ Используется в разделе 707.
⟨ Заверши эллипс, копируя противоположность уже расчитанной половины 537 ⟩
Используется в разделе 527.
⟨ Заверши, выбирая углы и назначая управляющие точки 297 ⟩ Используется в разделе 284.
⟨ Загрузи динамическую память 1195 ⟩ Используется в разделе 1187.
⟨ Загрузи константы для проверки целостности 1191 ⟩ Используется в разделе 1187.
⟨ Загрузи несколько вещей и завершающее контрольное слово 1199 ⟩ Используется в разделе 1187.
⟨ Загрузи пул строк 1193 ⟩ Используется в разделе 1187.
⟨ Загрузи таблицу эквивалентов и хеш-таблицу 1197 ⟩ Используется в разделе 1187.
⟨ Займись избыточным или несовместным уравнением 1008 ⟩ Используется в разделе 1006.
⟨ Заключительные процедуры 1205, 1209, 1210, 1212 ⟩ Используется в разделе 1202.
⟨ Закончи вычисления параметра *paint_row* вставкой завершающего перехода; **goto** *done*, если закраска не нужна 584 ⟩ Используется в разделе 582.
⟨ Закрой форматный файл 1201 ⟩ Используется в разделе 1186.
⟨ Замени a приближением к $\sqrt{a^2+b^2}$ 125 ⟩ Используется в разделе 124.
⟨ Замени a приближением с $\sqrt{a^2-b^2}$ 127 ⟩ Используется в разделе 126.
⟨ Замени *cur_sym*, если он в *subst_list* 686 ⟩ Используется в разделе 685.
⟨ Замени интервал значений их средним 1122 ⟩ Используется в разделе 1121.
⟨ Замени на ‘плохую переменную’ 701 ⟩ Используется в разделе 700.
⟨ Замени переменную x с *independent* на *dependent* или *known* 615 ⟩ Используется в разделе 610.
⟨ Замени предварительное перо 1063 ⟩ Используется в разделе 1062.
⟨ Замени узел q на путь для эллиптического пера 866 ⟩ Используется в разделе 865.
⟨ Запиши возможный переход в колонке m 583 ⟩ Используется в разделе 582.
⟨ Запиши линейный сегмент от (xx, yy) до (xp, yp) в *env_move* 516 ⟩ Используется в разделе 515.
⟨ Запиши линейный сегмент от (xx, yy) до (xp, yp) дважды в *env_move* 522 ⟩ Используется в разделе 521.
⟨ Запиши метку в подпрограмму лигатуры/керна и **goto** *continue* 1111 ⟩ Используется в разделе 1107.
⟨ Запиши новый максимальный коэффициент типа t 814 ⟩ Используется в разделе 812.
⟨ Заполни в управляющих данных между последовательными точками разрывлв p и q 278 ⟩
Используется в разделе 273.
⟨ Заполни в управляющих точках между p и следующей точкой разрыва, затем передвинь p на ту точку разрыва 273 ⟩ Используется в разделе 269.
⟨ Заткни брешь в *right_type*(pp), если возможно 889 ⟩ Используется в разделе 887.
⟨ Заткни брешь в *right_type*(q), если возможно 888 ⟩ Используется в разделе 887.
⟨ Заяви о делении на нуль 838 ⟩ Используется в разделе 837.
⟨ Заяви об излишнем уравнении 623 ⟩ Используется в разделах 622, 1004 и 1008.

⟨ Иди к следующей строке файла или **goto** *restart*, если нет следующей строки 679 ⟩
Используется в разделе 669.
⟨ Избавься от случаев $a < 0$ и/или $b > l$ 979 ⟩ Используется в разделе 978.
⟨ Извлеки из стека условий 745 ⟩ Используется в разделах 748, 749 и 751.
⟨ Измени знак текущего выражения 903 ⟩ Используется в разделе 898.
⟨ Измени известное известным 973 ⟩ Используется в разделе 970.
⟨ Или начни вызов макроса без суффикса, или готовься к вызову с суффиксом 845 ⟩
Используется в разделе 844.
⟨ Интерполируй новые вершины в структуре данных эллипса, пока возиожны улучшения 531 ⟩
Используется в разделе 527.
⟨ Искази TFM высóты, глубúны и курсивные поправки 1126 ⟩ Используется в разделе 1206.
⟨ Искази TFM ширúны 1124 ⟩ Используется в разделе 1206.
⟨ Используй бисекцию, чтобы найти точку пересечения, если она существует 392 ⟩
Используется в разделе 391.
⟨ Исправь значения α, β, γ, если нужно, чтобы не получалось вырожденных линий нулевой длины 529 ⟩ Используется в разделе 528.
⟨ Исправь код октанта в сегментах с уменьшающимся y 418 ⟩ Используется в разделе 413.
⟨ Исправь поля перехода и подстрой число поворотов 459 ⟩ Используется в разделе 452.
⟨ Ищи в *eqtb* эквиваленты, равные p 209 ⟩ Используется в разделе 185.
⟨ Ищи нижний индекс; замени *cur_cmd* лексемой *numeric_token*, если найдена 846 ⟩
Используется в разделе 844.
⟨ Ищи первую точку разрыва h на пути; вставь искусственную точку разрыва, если путь — неразорванный цикл 272 ⟩ Используется в разделе 269.
⟨ Константы во внешнем блоке 11 ⟩ Используется в разделе 4.
⟨ Кончи зачернять в (m, n) 1171 ⟩ Используется в разделе 1169.
⟨ Копируй большой узел p 857 ⟩ Используется в разделе 855.
⟨ Копируй нескошенные и неповёрнутые координаты узла *ww* 485 ⟩ Используется в разделе 484.
⟨ Копируй оба *sorted* и *unsorted* списка p в *pp* 335 ⟩ Используется в разделах 334 и 341.
⟨ Локальне переменные для начальной настройки 19, 130 ⟩ Используется в разделе 4.
⟨ Масштабируй границы, сдвинь их и **return** 964 ⟩ Используется в разделе 963.
⟨ Масштабируй координаты x каждой строки на s 343 ⟩ Используется в разделе 342.
⟨ Местные переменные для расчётов форматирования 641 ⟩ Используется в разделе 635.
⟨ Метки во внешнем блоке 6 ⟩ Используется в разделе 4.
⟨ Найди индекс k, такой что $s_{k-1} \leqslant dy/dx < s_k$ 502 ⟩ Используется в разделе 494.
⟨ Найди наименьшее *lk_offset* и подстрой все остатки 1138 ⟩ Используется в разделе 1137.
⟨ Найди начальную точку, f 399 ⟩ Используется в разделе 398.
⟨ Найди начальный наклон, dy/dx 501 ⟩ Используется в разделе 494.
⟨ Найди тип аппроксимации *tt* и соответствующий q 850 ⟩ Используется в разделе 844.
⟨ Найди узел q в списке p, коэффициент v которого наибольший 611 ⟩ Используется в разделе 610.
⟨ Направь аргументы и замещающий текст в считыватель 736 ⟩ Используется в разделе 720.
⟨ Настрой *selector* печати, основанный на *interaction* 70 ⟩ Используется в разделах 1023 и 1211.
⟨ Настрой заголовок, чтобы отразить новые границы 364 ⟩ Используется в разделе 354.
⟨ Настрой массив заголовков новых списков границ 356 ⟩ Используется в разделе 354.
⟨ Настрой процедуры ввода 657, 660 ⟩ Используется в разделе 1211.
⟨ Настрой процедуры вывода 55, 61, 783, 792 ⟩ Используется в разделе 1204.
⟨ Настрой случайное зерно в *cur_exp* 1022 ⟩ Используется в разделе 1021.
⟨ Настрой структуру данных эллипса, начиная с направлений $(0, -1)$, $(1, 0)$, $(0, 1)$ 528 ⟩
Используется в разделе 527.
⟨ Настрой таблицы (делает только INIMF) 176, 193, 203, 229, 324, 475, 587, 702, 759, 911, 1116, 1127, 1185 ⟩
Используется в разделе 1210.
⟨ Настройся для вычислений показа 581 ⟩ Используется в разделе 577.
⟨ Настройся для перемещений двойных огибающих 519 ⟩ Используется в разделе 518.

⟨ Настройся для перемещений одинарных огибающих 513 ⟩ Используется в разделе 512.
⟨ Настройся для пересечений на нулевом уровне 558 ⟩ Используется в разделе 556.
⟨ Начни вывод с частью описателя; **return** в случае капсулы 237 ⟩ Используется в разделе 235.
⟨ Начни зачернять в (m,n) 1170 ⟩ Используется в разделе 1169.
⟨ Начни или заверши ввод из файла 711 ⟩ Используется в разделе 707.
⟨ Начни линейные уравнения; или **return** с управляющими точками на своих местах, если линейные уравнения не нужно решать 285 ⟩ Используется в разделе 284.
⟨ Начни новую строку в (m,n) 1172 ⟩ Используется в разделе 1170.
⟨ Нормируй данное направление для лучшей точности; но **return** с нулевым итогом, если оно — нуль 540 ⟩ Используется в разделе 539.
⟨ Нумерованные случаи для *debug_help* 1213 ⟩ Используется в разделе 1212.
⟨ Обеспечь диагностические данные, если запрашивается 825 ⟩ Используется в разделе 823.
⟨ Обеспечь, чтобы $type(p) = proto\_dependent$ 969 ⟩ Используется в разделе 968.
⟨ Обменя координаты x и y кривых между p и q 423 ⟩ Используется в разделе 420.
⟨ Обнови наибольшую и наименьшую величи́ны 351 ⟩ Используется в разделе 349.
⟨ Обработай знаки в кавычках: `#@`, `@` или `@#` 690 ⟩ Используется в разделе 685.
⟨ Обработай квадратный корень от нуля или отрицательного аргумента 122 ⟩ Используется в разделе 121.
⟨ Обработай необычные случаи, притворяющиеся переменными, и **goto** *restart* или **goto** *done*, если уместно; иначе копируй переменную и **goto** *done* 852 ⟩ Используется в разделе 844.
⟨ Обработай неопределённый аргумент 140 ⟩ Используется в разделе 139.
⟨ Обработай неположительный логарифм 134 ⟩ Используется в разделе 132.
⟨ Обработай особый случай бесконечного наклона 505 ⟩ Используется в разделе 494.
⟨ Обработай особый случай длины 1 и **goto** *found* 206 ⟩ Используется в разделе 205.
⟨ Обработай ошибочный *pyth_sub* и установи $a \leftarrow 0$ 128 ⟩ Используется в разделе 126.
⟨ Общие переменные 13, 20, 25, 29, 31, 38, 42, 50, 54, 68, 71, 74, 91, 97, 129, 137, 144, 148, 159, 160, 161, 166, 178, 190, 196, 198, 200, 201, 225, 230, 250, 267, 279, 283, 298, 308, 309, 327, 371, 379, 389, 395, 403, 427, 430, 448, 455, 461, 464, 507, 552, 555, 557, 566, 569, 572, 579, 585, 592, 624, 628, 631, 633, 634, 659, 680, 699, 738, 752, 767, 768, 775, 782, 785, 791, 796, 813, 821, 954, 1077, 1084, 1087, 1096, 1119, 1125, 1130, 1149, 1152, 1162, 1183, 1188, 1203 ⟩ Используется в разделе 4.
⟨ Объедини динамическую память в один большой свободный узел 1207 ⟩ Используется в разделе 1206.
⟨ Объяви о невыполнимости равенства 1002 ⟩ Используется в разделе 1001.
⟨ Объяви основные подпрограммы для списков зависимостей 594, 600, 602, 603, 604 ⟩ Используется в разделе 246.
⟨ Объяви основные процедуры разбора 823, 860, 862, 864, 868, 892 ⟩ Используется в разделе 1202.
⟨ Объяви подпрограммы для печати выражений 257, 332, 388, 473, 589, 801, 807 ⟩ Используется в разделе 246.
⟨ Объяви подпрограммы для повторного использования памяти 268, 385, 487, 620, 809 ⟩ Используется в разделе 246.
⟨ Объяви подпрограммы, нужные для *big_trans* 968, 971, 972, 974 ⟩ Используется в разделе 966.
⟨ Объяви подпрограммы, нужные для *make_exp_copy* 856, 858 ⟩ Используется в разделе 855.
⟨ Объяви подпрограммы, нужные для *make_spec* 405, 406, 419, 426, 429, 431, 432, 433, 440, 451 ⟩ Используется в разделе 402.
⟨ Объяви подпрограммы, нужные для *offset_prep* 493, 497 ⟩ Используется в разделе 491.
⟨ Объяви подпрограммы, нужные для *solve_choices* 296, 299 ⟩ Используется в разделе 284.
⟨ Объяви процедуру *check_delimiter* 1032 ⟩ Используется в разделе 697.
⟨ Объяви процедуру *dep_finish* 935 ⟩ Используется в разделе 930.
⟨ Объяви процедуру *dual_moves* 518 ⟩ Используется в разделе 506.
⟨ Объяви процедуру *flush_below_variable* 247 ⟩ Используется в разделе 246.
⟨ Объяви процедуру *flush_cur_exp* 808, 820 ⟩ Используется в разделе 246.
⟨ Объяви процедуру *flush_string* 43 ⟩ Используется в разделе 73.
⟨ Объяви процедуру *known_pair* 872 ⟩ Используется в разделе 871.
⟨ Объяви процедуру *macro_call* 720 ⟩ Используется в разделе 706.

⟨ Объяви процедуру *make_eq* 1001 ⟩ Используется в разделе 995.
⟨ Объяви процедуру *make_exp_copy* 855 ⟩ Используется в разделе 651.
⟨ Объяви процедуру *print_arg* 723 ⟩ Используется в разделе 720.
⟨ Объяви процедуру *print_cmd_mod* 625 ⟩ Используется в разделе 227.
⟨ Объяви процедуру *print_dp* 805 ⟩ Используется в разделе 801.
⟨ Объяви процедуру *print_macro_name* 722 ⟩ Используется в разделе 720.
⟨ Объяви процедуру *print_weight* 333 ⟩ Используется в разделе 332.
⟨ Объяви процедуру *runaway* 665 ⟩ Используется в разделе 162.
⟨ Объяви процедуру *scan_text_arg* 730 ⟩ Используется в разделе 720.
⟨ Объяви процедуру *show_token_list* 217 ⟩ Используется в разделе 162.
⟨ Объяви процедуру *skew_line_edges* 510 ⟩ Используется в разделе 506.
⟨ Объяви процедуру *solve_choices* 284 ⟩ Используется в разделе 269.
⟨ Объяви процедуру *split_cubic* 410 ⟩ Используется в разделе 406.
⟨ Объяви процедуру *try_eq* 1006 ⟩ Используется в разделе 995.
⟨ Объяви процедуры вывода файла общего шрифта 1154, 1155, 1157, 1158, 1159, 1160, 1161, 1163, 1165 ⟩ Используется в разделе 989.
⟨ Объяви процедуры для бинарных операций 923, 928, 930, 943, 946, 949, 953, 960, 961, 962, 963, 966, 976, 977, 978, 982, 984, 985 ⟩ Используется в разделе 922.
⟨ Объяви процедуры для унарных операций 899, 900, 901, 904, 908, 910, 913, 916, 919 ⟩ Используется в разделе 898.
⟨ Объяви прячущие/достающие подпрограммы 799, 800 ⟩ Используется в разделе 801.
⟨ Объяви рабочие процедуры для использования *do_statement* 995, 996, 1015, 1021, 1029, 1031, 1034, 1035, 1036, 1040, 1041, 1044, 1045, 1046, 1049, 1050, 1051, 1054, 1057, 1059, 1070, 1071, 1072, 1073, 1074, 1082, 1103, 1104, 1106, 1177, 1186 ⟩ Используется в разделе 989.
⟨ Объяви различные процедуры, заявленные как *forward* 224 ⟩ Используется в разделе 1202.
⟨ Объяви функцию *open_base_file* 779 ⟩ Используется в разделе 1187.
⟨ Объяви функцию *scan_declared_variable* 1011 ⟩ Используется в разделе 697.
⟨ Объяви функцию *tfm_check* 1098 ⟩ Используется в разделе 1070.
⟨ Объяви функцию *trivial_knot* 486 ⟩ Используется в разделе 484.
⟨ Определи границу октанта q, предшествующую f 400 ⟩ Используется в разделе 398.
⟨ Определи значения до-и-после обоих координат 445 ⟩ Используется в разделах 444 и 446.
⟨ Определи код октанта для направления (dx, dy) 480 ⟩ Используется в разделе 479.
⟨ Определи набор команд расширения 1113 ⟩ Используется в разделе 1106.
⟨ Определи натяжение и/или управляющие точки 881 ⟩ Используется в разделе 874.
⟨ Определи начальную $(m0, n0)$ и конечную $(m1, n1)$ точки решётки огибающей 508 ⟩ Используется в разделе 506.
⟨ Определи начальную и конечную точки решётки $(m0, n0)$ и $(m1, n1)$ 467 ⟩ Используется в разделе 465.
⟨ Определи параметры объединения путей; но **goto** *finish_path*, если есть только направление 874 ⟩ Используется в разделе 869.
⟨ Определи расширение файла, *gf_ext* 1164 ⟩ Используется в разделе 1163.
⟨ Определи список зависимосей s для замены на независимую переменную p 816 ⟩ Используется в разделе 815.
⟨ Определи число n уже замещённых аргументов и установи *tail* на конец списка *arg_list* 724 ⟩ Используется в разделе 720.
⟨ Определи, выгружен ли символ 1181 ⟩ Используется в разделе 906.
⟨ Освободи большой узел 810 ⟩ Используется в разделе 809.
⟨ Освободи любые неучаствующие капсулы *independent* 925 ⟩ Используется в разделе 922.
⟨ Освободи независимую переменную 812 ⟩ Используется в разделе 809.
⟨ Освободи последние байты *gf_buf* 1156 ⟩ Используется в разделе 1182.
⟨ Освободи список зависимостей 811 ⟩ Используется в разделе 809.
⟨ Основные процедуры печати 57, 58, 59, 60, 62, 63, 64, 103, 104, 187, 195, 197, 773 ⟩ Используется в разделе 4.
⟨ Оставь команду краёв, потому что нет переменной 1060 ⟩ Используется в разделах 1059, 1070, 1071 и 1074.

⟨ Отбрось избыточные вхождения «граница-вес» из *sorted*(*p*) 349 ⟩ Используется в разделе 348.
⟨ Отбрось незаконные символы после объявленной переменной 1016 ⟩ Используется в разделе 1015.
⟨ Отбрось неразбираемый хлам, обнаруженный после оператора 991 ⟩ Используется в разделе 989.
⟨ Отдели другую *rising* кривую для *fin_offset_prep* 504 ⟩ Используется в разделе 503.
⟨ Отметь все узлы с кодом октанта, вычисли наибольшее смещение, и установи *hh* на узел, начинающий первый октант; **goto** *not_found*, если есть трудности 479 ⟩ Используется в разделе 477.
⟨ Отправь текущее выражение как название выходного файла 1179 ⟩ Используется в разделе 994.
⟨ Отрази данные границы и веса в *sorted*(*p*) 339 ⟩ Используется в разделе 337.
⟨ Отрази данные границы и веса в *unsorted*(*p*) 338 ⟩ Используется в разделе 337.
⟨ Отсеки значения всех координат, превышающие *max_allowed*, и пометь номера сегментов в каждом поле *left_type* 404 ⟩ Используется в разделе 402.
⟨ Отследи текущую бинарную операцию 924 ⟩ Используется в разделе 922.
⟨ Отследи текущую унарную операцию 902 ⟩ Используется в разделе 898.
⟨ Отслеживай начало цикла 762 ⟩ Используется в разделе 760.
⟨ Передай двойные перемещения из массива *move* в *env_move* 520 ⟩ Используется в разделе 518.
⟨ Передай перемещения из массива *move* в *env_move* 514 ⟩ Используется в разделе 512.
⟨ Передвинь *p* на узел *q*, удаляя любые «мёртвые» сплайны, которые могли быть введены процессом разбиения 492 ⟩ Используется в разделе 491.
⟨ Передвинь указатель *p* на следующую вертикальную границу после уничтожения предыдущей 360 ⟩ Используется в разделе 358.
⟨ Передвинь указатель *r* на следующую вертикальную границу 359 ⟩ Используется в разделе 358.
⟨ Перейди к следующей оставшейся тройке (p,q,r), удаляя и пропуская линии ненулеувой длины, которые могут присутствовать; **goto** *done*, если все тройки обработаны 532 ⟩ Используется в разделе 531.
⟨ Перейди к следующей паре (cur_t, cur_tt) 560 ⟩ Используется в разделе 556.
⟨ Перейди к строке *n0*, указанной *p* 377 ⟩ Используется в разделах 375, 376, 381, 382, 383 и 384.
⟨ Переключись на правый подинтервал 318 ⟩ Используется в разделе 317.
⟨ Перемести зависимую переменную *p* в обе части узла пары *r* 947 ⟩ Используется в разделе 946.
⟨ Переместись вверх на *n* шагов 315 ⟩ Используется в разделе 314.
⟨ Переместись вверх, затем вправо 320 ⟩ Используется в разделах 317 и 317.
⟨ Переместись вправо на *m* шагов 316 ⟩ Используется в разделе 314.
⟨ Переместись право, затем вверх 319 ⟩ Используется в разделах 317 и 317.
⟨ Печатай банер с датой и временем 790 ⟩ Используется в разделе 788.
⟨ Печатай вид списка лексем 638 ⟩ Используется в разделе 636.
⟨ Печатай данные для кривой, которая начинает *curl* или *given* 263 ⟩ Используется в разделе 258.
⟨ Печатай данные для кривой, которая начинает *open* 262 ⟩ Используется в разделе 258.
⟨ Печатай данные для смежных узловых точек *p* и *q* 258 ⟩ Используется в разделе 257.
⟨ Печатай две строки, используя ловкие псевдопечатанные сведения 643 ⟩ Используется в разделе 636.
⟨ Печатай две точки, за которыми следует *given* или *curl*, если есть 259 ⟩ Используется в разделе 257.
⟨ Печатай значение текущего цикла 639 ⟩ Используется в разделе 638.
⟨ Печатай имя макроса, определённого **vardef** 640 ⟩ Используется в разделе 638.
⟨ Печатай коэффициент, если он не равен ± 1.0 590 ⟩ Используется в разделе 589.
⟨ Печатай кубический сплайн между *p* и *q* 397 ⟩ Используется в разделе 394.
⟨ Печатай натяжение между *p* и *q* 260 ⟩ Используется в разделе 258.
⟨ Печатай недавно занятые позиции 184 ⟩ Используется в разделе 180.
⟨ Печатай нескошенные и неповёрнутые координаты узла *ww* 474 ⟩ Используется в разделе 473.
⟨ Печатай перечень возможных действий 80 ⟩ Используется в разделе 79.
⟨ Печатай повороты, начинающиеся на *q*, если есть, и передвинь *q* 401 ⟩ Используется в разделах 398 и 398.
⟨ Печатай позицию текущей строки 637 ⟩ Используется в разделе 636.
⟨ Печатай сокращённое значение *v* в формате, определяемом *t* 802 ⟩ Используется в разделе 801.
⟨ Печатай справку и **goto** *continue* 84 ⟩ Используется в разделе 79.

⟨ Печатай строку *cur_exp*, как сообщение об ошибке 1086 ⟩ Используется в разделе 1082.
⟨ Печатай строку *err_help*, возможно, на нескольких строках экрана 85 ⟩ Используется в разделах 84 и 86.
⟨ Печатай строку r, как символьную лексему и установи c на её класс 223 ⟩ Используется в разделе 218.
⟨ Печатай строку дигностических данных, чтобы ввести этот октант 509 ⟩ Используется в разделе 508.
⟨ Печатай управляющие точки между p и q, затем **goto** *done1* 261 ⟩ Используется в разделе 258.
⟨ Побеспокойся о плохом операторе 990 ⟩ Используется в разделе 989.
⟨ Поверни сплайн между p и q; затем **goto** *found*, если повёрнутый сплайн идёт прямо на восток в некоторое время tt; но **goto** *not_found*, если пройден полный циклический путь 541 ⟩
Используется в разделе 539.
⟨ Повтори каждую строку ровно s раз 341 ⟩ Используется в разделе 340.
⟨ Повторяй цикл 712 ⟩ Используется в разделе 707.
⟨ Погрузись в структуру 1047 ⟩ Используется в разделе 1046.
⟨ Подготовься и переключись на подходящий случай, основываясь на *octant* 380 ⟩
Используется в разделе 378.
⟨ Поднимись на один уровень, выталкивая лексему в список q и заменяя p его предком 236 ⟩
Используется в разделе 235.
⟨ Подправь баланс для неразграниченного аргумента; **goto** *done*, если сделано 732 ⟩
Используется в разделе 730.
⟨ Подправь баланс для разграниченного аргумента; **goto** *done*, если сделано 731 ⟩
Используется в разделе 730.
⟨ Подправь баланс; если он нулевой, **goto** *done* 687 ⟩ Используется в разделе 685.
⟨ Подправь данные h, чтобы вычислить разницу смещений 367 ⟩ Используется в разделе 366.
⟨ Подправь координаты $(r0, c0)$ и $(r1, c1)$, чтобы они лежали в должной области 575 ⟩
Используется в разделе 574.
⟨ Подставь новые зависимости вместо p 818 ⟩ Используется в разделе 815.
⟨ Подставь новые протозависимости вместо p 819 ⟩ Используется в разделе 815.
⟨ Подстрой θ_n к равному θ_0 и **goto** *found* 291 ⟩ Используется в разделе 287.
⟨ Пожалуйся на конфликт ярлыка символа 1105 ⟩ Используется в разделе 1104.
⟨ Пожалуйся на неверную особое действие [special operation] 1178 ⟩ Используется в разделе 1177.
⟨ Пожалуйся на негодный тип 1055 ⟩ Используется в разделе 1054.
⟨ Пожалуйся на нециклический путь и **goto** *not_found* 1067 ⟩ Используется в разделе 1064.
⟨ Пожалуйся на плохой путь пера 478 ⟩ Используется в разделе 477.
⟨ Покажи большой узел 803 ⟩ Используется в разделе 802.
⟨ Покажи булево значение *cur_exp* 750 ⟩ Используется в разделе 748.
⟨ Покажи двусловную лексему 219 ⟩ Используется в разделе 218.
⟨ Покажи лексему p и установи c на её класс; но **return**, если есть трудности 218 ⟩
Используется в разделе 217.
⟨ Покажи лексему параметра 222 ⟩ Используется в разделе 218.
⟨ Покажи макрос переменной 1048 ⟩ Используется в разделе 1046.
⟨ Покажи новую зависимость 613 ⟩ Используется в разделе 610.
⟨ Покажи объявленную, но не определённую переменную 806 ⟩ Используется в разделе 802.
⟨ Покажи преобразованную зависимость 817 ⟩ Используется в разделе 816.
⟨ Покажи сложный тип 804 ⟩ Используется в разделе 802.
⟨ Покажи совакупный индекс 221 ⟩ Используется в разделе 218.
⟨ Покажи текст раскрываемого макро и существующих аргументов 721 ⟩ Используется в разделе 720.
⟨ Покажи текущий контекст 636 ⟩ Используется в разделе 635.
⟨ Покажи точки строки p границы в строке r экрана 578 ⟩ Используется в разделе 577.
⟨ Покажи числовую или строковую лексему или лексему капсулы 1042 ⟩ Используется в разделе 1041.
⟨ Покажи числовую лексему 220 ⟩ Используется в разделе 219.
⟨ Положи левую квадратную скобку и выражение назад для повторного считывания 847 ⟩
Используется в разделах 846 и 859.
⟨ Получи данные направления, разделённые запятыми 878 ⟩ Используется в разделе 877.

⟨ Получи дробную часть f числовой лексемы 674 ⟩ Используется в разделе 669.
⟨ Получи первую строку ввода и готовься начать 1211 ⟩ Используется в разделе 1204.
⟨ Получи сохранённую числовую или строковую лексему, или лексему капсулы и **return** 678 ⟩ Используется в разделе 676.
⟨ Получи строковую лексему и **return** 671 ⟩ Используется в разделе 669.
⟨ Получи целую часть n числовой лексемы; установи $f \leftarrow 0$ и **goto** *fin_numeric_token*, если нет десятичной точки 673 ⟩ Используется в разделе 669.
⟨ Помести в стек условий 744 ⟩ Используется в разделе 748.
⟨ Помести желаемое имя файла в (*cur_name*, *cur_ext*, *cur_area*) 795 ⟩ Используется в разделе 793.
⟨ Помести каждый примитив METAFONT'а в хеш-таблицу 192, 211, 683, 688, 695, 709, 740, 893, 1013, 1018, 1024, 1027, 1037, 1052, 1079, 1101, 1108, 1176 ⟩ Используется в разделе 1210.
⟨ Помести сведения о направлении до объединения в узел q 879 ⟩ Используется в разделе 874.
⟨ Помести сведения о направлении после объединения в x и t 880 ⟩ Используется в разделе 874.
⟨ Помести справку в log-файл 86 ⟩ Используется в разделе 77.
⟨ Помести строку во входной буфер 716 ⟩ Используется в разделе 707.
⟨ Помести текущее преобразование в *cur_exp* 955 ⟩ Используется в разделе 953.
⟨ Попробуй получить другое имя log-файла 789 ⟩ Используется в разделе 788.
⟨ Попробуй разместить в узле p и в физически следующих за ним узлах и **goto** *found*, если размещение оказалось возможным 169 ⟩ Используется в разделе 167.
⟨ Поругай недопустимый символ и **goto** *restart* 670 ⟩ Используется в разделе 669.
⟨ Поругай пользователя за лишнее **endfor** 708 ⟩ Используется в разделе 707.
⟨ Поругай пропущенный разделитель строки и **goto** *restart* 672 ⟩ Используется в разделе 671.
⟨ Построй путь от *pp* до *qq* длины $\lceil b \rceil$ 980 ⟩ Используется в разделе 978.
⟨ Построй путь от *pp* до *qq* нулевой длины 981 ⟩ Используется в разделе 978.
⟨ Построй список смещений для k-ого октанта 481 ⟩ Используется в разделе 477.
⟨ Пошли ненулевые смещения в выходной файл 1166 ⟩ Используется в разделе 1165.
⟨ Преврати одноточечные пути в мёртвые циклы 563 ⟩ Используется в разделе 562.
⟨ Преобразуй (x, y) к определяемому величиной q 146 ⟩ Используется в разделе 145.
⟨ Преобразуй известный большой узел 970 ⟩ Используется в разделе 966.
⟨ Преобразуй координаты x 436 ⟩ Используется в разделе 433.
⟨ Преобразуй координаты y 439 ⟩ Используется в разделе 433.
⟨ Преобразуй косоугольные координаты 444 ⟩ Используется в разделе 440.
⟨ Преобразуй левый операнд, p, в неполный путь, оканчивающийся на q; но **return**, если p имеет негодный тип 870 ⟩ Используется в разделе 869.
⟨ Преобразуй неизвестный большой узел и **return** 967 ⟩ Используется в разделе 966.
⟨ Преобразуй правый операнд *cur_exp* в неполный путь от *pp* до *qq* 885 ⟩ Используется в разделе 869.
⟨ Преобразуй суффикс в строку 840 ⟩ Используется в разделе 823.
⟨ Прибавь или вычти текущее выражение из p 929 ⟩ Используется в разделе 922.
⟨ Припрячь независимое *cur_exp* в большом узле 829 ⟩ Используется в разделе 827.
⟨ Присвой текущее выражение внутренней переменной 999 ⟩ Используется в разделе 996.
⟨ Присвой текущее выражение переменной *lhs* 1000 ⟩ Используется в разделе 996.
⟨ Присоедини замещаемый текст к концу узла p 698 ⟩ Используется в разделе 697.
⟨ Притворись, что мы читаем новый однострочный файл 717 ⟩ Используется в разделе 716.
⟨ Проверь значения «констант» на совместимость 14, 154, 204, 214, 310, 553, 777 ⟩ Используется в разделе 1204.
⟨ Проверь контрольную сумму пула строк 53 ⟩ Используется в разделе 52.
⟨ Проверь места, где $B(y_1, y_2, y_3; t) = 0$, выполняется ли $B(x_1, x_2, x_3; t) \geq 0$ 547 ⟩ Используется в разделе 546.
⟨ Проверь на восточное направление, когда $y_1 y_3 = y_2^2$; или **goto** *found*, или **goto** *done* 548 ⟩ Используется в разделе 546.
⟨ Проверь наличие двоеточия 756 ⟩ Используется в разделе 755.
⟨ Проверь однословный список *avail* 181 ⟩ Используется в разделе 180.

⟨ Проверь округлённое число 1068 ⟩ Используется в разделе 1064.
⟨ Проверь предварительный вес 1056 ⟩ Используется в разделе 1054.
⟨ Проверь список *avail* переменного размера 182 ⟩ Используется в разделе 180.
⟨ Проверь список линейных зависимостей 617 ⟩ Используется в разделе 180.
⟨ Проверь флаги недоступных узлов 183 ⟩ Используется в разделе 180.
⟨ Проверь, все ли диагональные округления безопасны 446 ⟩ Используется в разделе 444.
⟨ Проверь, оба ли узла p и pp типа *structured* 243 ⟩ Используется в разделе 242.
⟨ Проверь, приравнены ли неизвестные 938 ⟩ Используется в разделе 936.
⟨ Проверь, что надлежащий правый разделитель присутствует 727 ⟩ Используется в разделе 726.
⟨ Проверь, что обе части x и y величины p известны; помести их в *cur_x* и *cur_y* 873 ⟩
Используется в разделе 872.
⟨ Проверь, что текущее выражение суть допустимое значение натяжения 883 ⟩
Используется в разделах 882 и 882.
⟨ Пройди по списку зависимостей для переменной t, исправляя все узлы, и заканчивая последней ссылкой q 605 ⟩ Используется в разделе 604.
⟨ Пропусти вниз $prev\_n - n$ строк 1174 ⟩ Используется в разделе 1172.
⟨ Пропусти до **elseif**, или **else**, или **fi**, затем **goto** *done* 749 ⟩ Используется в разделе 748.
⟨ Пропусти до колонки m в следующей строке и **goto** *done* или пропусти нуль строк 1173 ⟩
Используется в разделе 1172.
⟨ Проследи дробное умножение 945 ⟩ Используется в разделе 944.
⟨ Проследи текущее присвоение 998 ⟩ Используется в разделе 996.
⟨ Проследи текущее уравнение 997 ⟩ Используется в разделе 995.
⟨ Процедуры обработки ошибок 73, 76, 77, 88, 89, 90 ⟩ Используется в разделе 4.
⟨ Псевдопечатай список лексем 645 ⟩ Используется в разделе 636.
⟨ Псевдопечатай строку 644 ⟩ Используется в разделе 636.
⟨ Разбей все сплайны между p и q, чтобы итоги проходили к первому квадранту; но **return** или **goto** *continue*, если сплайн от p до q мёртв 413 ⟩ Используется в разделе 406.
⟨ Разбей для нового уровня пересечения 559 ⟩ Используется в разделе 556.
⟨ Разбей кубическую кривую второй раз по x' 412 ⟩ Используется в разделе 411.
⟨ Разбей кубическую кривую второй раз по $x' - y'$ 425 ⟩ Используется в разделе 424.
⟨ Разбей кубическую кривую второй раз по y' 416 ⟩ Используется в разделе 415.
⟨ Разбей кубическую кривую по x', возможно дважды 411 ⟩ Используется в разделе 407.
⟨ Разбей кубическую кривую по $x' - y'$, возможно дважды 424 ⟩ Используется в разделе 420.
⟨ Разбей кубическую кривую по y', возможно дважды 415 ⟩ Используется в разделе 413.
⟨ Разбей сплайн в t, и отдели другой сплайн, если производная переходит назад 499 ⟩
Используется в разделе 497.
⟨ Разбей сплайн между p и q, если нужно, на сплайны, связанные с отдельными смещениями, после этого q должен указывать на конец последнего такого сплайна 494 ⟩ Используется в разделе 491.
⟨ Разбей сплайн между p и q, чтобы итоги проходили к первому октанту 420 ⟩
Используется в разделе 419.
⟨ Разбей сплайн между p и q, чтобы итоги проходили к правой полуплоскости 407 ⟩
Используется в разделе 406.
⟨ Раздели переменные на два, чтобы избежать трудности переполнения 313 ⟩ Используется в разделе 311.
⟨ Раздели список p на 2^n 616 ⟩ Используется в разделе 615.
⟨ Раздели список p на $-v$, удаляя узел q 612 ⟩ Используется в разделе 610.
⟨ Раскрой эту лексему после следующей 715 ⟩ Используется в разделе 707.
⟨ Реши: идти по часовой стрелке или против 454 ⟩ Используется в разделе 452.
⟨ Сведи к простому случаю двух данных и **return** 301 ⟩ Используется в разделе 285.
⟨ Сведи к простому случаю прямой линии и **return** 302 ⟩ Используется в разделе 285.
⟨ Сведи к случаю $a, c \geqslant 0$ и $b, d > 0$ 118 ⟩ Используется в разделе 117.
⟨ Сведи к случаю $f \geqslant 0$ и $q > 0$ 110 ⟩ Используется в разделах 109 и 112.
⟨ Сведи сравнение больших узлов к сравнению скалярных 939 ⟩ Используется в разделе 936.

⟨ Свяжи новый узел атрибута r вместо узла p 241 ⟩ Используется в разделе 239.
⟨ Свяжи новый узел индекса r вместо узла p 240 ⟩ Используется в разделе 239.
⟨ Свяжи узел r с предыдущим узлом 482 ⟩ Используется в разделе 481.
⟨ Сдвинь границы на округлённые (tx, ty) 965 ⟩ Используется в разделе 964.
⟨ Сдвинь координаты пути q 867 ⟩ Используется в разделе 866.
⟨ Сделай заурядный одноточечный циклический путь 1066 ⟩ Используется в разделе 1065.
⟨ Сделай магический расчёт 646 ⟩ Используется в разделе 217.
⟨ Сделай особый узел узловой точки для **pencircle** 896 ⟩ Используется в разделе 895.
⟨ Сделай переменную $q + s$ вновь независимой 586 ⟩ Используется в разделе 232.
⟨ Сделай перемещения огибающей для текущего октанта и вставь их в данные точек 512 ⟩
Используется в разделе 506.
⟨ Сделай по одному перемещению каждого вида 317 ⟩ Используется в разделе 314.
⟨ Символ k не может печататься 49 ⟩ Используется в разделе 48.
⟨ Скажи пользователю об утере контроля и пробуй оправиться 663 ⟩ Используется в разделе 661.
⟨ Случаи *print_cmd_mod* для символьной печати примитивов 212, 684, 689, 696, 710, 741, 894, 1014, 1019, 1025, 1028, 1038, 1043, 1053, 1080, 1102, 1109, 1180 ⟩ Используется в разделе 625.
⟨ Случаи в *do_statement*, запускающей требуемые команды 1020, 1023, 1026, 1030, 1033, 1039, 1058, 1069, 1076, 1081, 1100, 1175 ⟩ Используется в разделе 992.
⟨ Смени уровень взаимодействия и **return** 81 ⟩ Используется в разделе 79.
⟨ Собери лексемы параметров для типа *base* 704 ⟩ Используется в разделе 703.
⟨ Собери неразграниченные параметры, помещая их в список r 705 ⟩ Используется в разделе 697.
⟨ Собери разграниченные параметры, помещая их в списки q и r 703 ⟩ Используется в разделе 697.
⟨ Собери числовую и дробную части числовой лексемы и **return** 675 ⟩ Используется в разделе 669.
⟨ Соедини неполные пути и сбрось p и q начало и конец итога 887 ⟩ Используется в разделе 869.
⟨ Создай *base_ident*, открой форматный файл и сообщи пользователю, что выгрузка началась 1200 ⟩
Используется в разделе 1186.
⟨ Создай первые 256 строк 48 ⟩ Используется в разделе 47.
⟨ Сообщи о неожиданном затруднении при выборе 270 ⟩ Используется в разделе 269.
⟨ Сообщи о переполнении входного буфера и прервись 34 ⟩ Используется в разделе 30.
⟨ Сообщи об избыточном или несовместном уравнении и **goto** *done* 1004 ⟩ Используется в разделе 1003.
⟨ Сортируй p в списке, начинающемся на *rover*, и перемещай p по *rlink*(p) 174 ⟩
Используется в разделе 173.
⟨ Сохрани сведения о ширине для кода символа c 1099 ⟩ Используется в разделе 1070.
⟨ Сохрани список байтов заголовка 1114 ⟩ Используется в разделе 1106.
⟨ Сохрани список размеров шрифта 1115 ⟩ Используется в разделе 1106.
⟨ Сохрани список шагов лигатуры/керна 1107 ⟩ Используется в разделе 1106.
⟨ Сохрани строку *cur_exp*, как *err_help* 1083 ⟩ Используется в разделе 1082.
⟨ Спустись на один уровень для атрибута *info*(t) 245 ⟩ Используется в разделе 242.
⟨ Спустись на один уровень для индекса *value*(t) 244 ⟩ Используется в разделе 242.
⟨ Спустись на предыдущий уровень и **goto** *not_found* 561 ⟩ Используется в разделе 560.
⟨ Спустись после группового индекса 1012 ⟩ Используется в разделе 1011.
⟨ Сравни текущее выражение с нулём 937 ⟩ Используется в разделе 936.
⟨ Срасти независимые пути вместе 890 ⟩ Используется в разделе 887.
⟨ Считай аргумент, представленный *info*(r) 729 ⟩ Используется в разделе 726.
⟨ Считай внутреннюю числовую величину 841 ⟩ Используется в разделе 823.
⟨ Считай вторую часть пары числовых 830 ⟩ Используется в разделе 826.
⟨ Считай выражение, за которым следует '**of** ⟨primary⟩' 734 ⟩ Используется в разделе 733.
⟨ Считай данное направление 877 ⟩ Используется в разделе 875.
⟨ Считай двуместную операцию с '**of**' между её операндами 839 ⟩ Используется в разделе 823.
⟨ Считай значения для использования в цикле 764 ⟩ Используется в разделе 755.
⟨ Считай имя файла в буфере 787 ⟩ Используется в разделе 786.

⟨ Считай индекс в квадратных скобках и установи *cur_cmd* ← *numeric_token* 861 ⟩
Используется в разделе 860.
⟨ Считай лексему или переменную, чтобы определить; установи n, *scanner_status* и *warning_info* 700 ⟩
Используется в разделе 697.
⟨ Считай неразграниченный аргумент (или аргументы) 733 ⟩ Используется в разделе 725.
⟨ Считай нуль-арную операцию 834 ⟩ Используется в разделе 823.
⟨ Считай одноместную операцию 835 ⟩ Используется в разделе 823.
⟨ Считай операцию построения пути; но **return**, если p неверного типа 869 ⟩ Используется в разделе 868.
⟨ Считай описание изгиба 876 ⟩ Используется в разделе 875.
⟨ Считай оставшиеся аргументы, если есть; установи r на первую лексему замещающего текста 725 ⟩
Используется в разделе 720.
⟨ Считай первичное, начинающееся с числовой лексемы 837 ⟩ Используется в разделе 823.
⟨ Считай первичное, ограниченное группой 832 ⟩ Используется в разделе 823.
⟨ Считай первичное, ограниченное разделителем 826 ⟩ Используется в разделе 823.
⟨ Считай первичную переменную; **goto** *restart*, если она оказалась макросом 844 ⟩
Используется в разделе 823.
⟨ Считай промежуточную конструкцию 859 ⟩ Используется в разделе 823.
⟨ Считай разграниченный аргумент, представленный полем *info*(r) 726 ⟩ Используется в разделе 725.
⟨ Считай строковую константу 833 ⟩ Используется в разделе 823.
⟨ Считай суффикс с необязательными разделителями 735 ⟩ Используется в разделе 733.
⟨ Считай текст цикла и помести его в стек управления циклами 758 ⟩ Используется в разделе 755.
⟨ Типы во внешнем блоке 18, 24, 37, 101, 105, 106, 156, 186, 565, 571, 627, 1151 ⟩ Используется в разделе 4.
⟨ Транслируй команду лигатуры/керна 1112 ⟩ Используется в разделе 1107.
⟨ Убери подзадачу для *make_moves* из стека 312 ⟩ Используется в разделе 311.
⟨ Убери типы *open* в точках разрыва 282 ⟩ Используется в разделе 278.
⟨ Увеличивай k, пока x не может умножаться на коэффициент 2^{-k}, и подправь y соответственно 133 ⟩
Используется в разделе 132.
⟨ Увеличь *del1*, *del2* и *del3* для большей точности; также ставь *del* на первый ненулевой элемент (*del1*, *del2*, *del3*) 408 ⟩ Используется в разделах 407, 413 и 420.
⟨ Увеличь z на аргумент (x, y) 143 ⟩ Используется в разделе 142.
⟨ Увеличь ещё память переменного размера и **goto** *restart* 168 ⟩ Используется в разделе 167.
⟨ Увеличь и уменьши $move[k-1]$ и $move[k]$ на δ_k 322 ⟩ Используется в разделе 321.
⟨ Удали $c -$ "0" лексем и **goto** *continue* 83 ⟩ Используется в разделе 79.
⟨ Удали все заголовки строк 353 ⟩ Используется в разделе 352.
⟨ Удали левый операнд из его контейнера, сделай отрицательным, и положи в список зависимостей p с постоянным членом q 1007 ⟩ Используется в разделе 1006.
⟨ Удали линию от p до q, и настрой вершину q, чтобы ввести новую линию 534 ⟩
Используется в разделе 531.
⟨ Удали мёртвые кубические сплайны 447 ⟩ Используется в разделе 402.
⟨ Удали пустые строки сверху и снизу; обнови граничные значения в заголовке 352 ⟩
Используется в разделе 348.
⟨ Удвой путь 1065 ⟩ Используется в разделе 1064.
⟨ Уменьши k на 1, сохраняя соотношения между x, y и q неизменными 123 ⟩ Используется в разделе 121.
⟨ Уменьши скорости, если нужно, чтобы остаться в ограничивающем треугольнике 300 ⟩
Используется в разделе 299.
⟨ Уменьши счётчик ссылок строк, если текущая лексема суть строка 743 ⟩
Используется в разделах 83, 742, 991 и 1016.
⟨ Умножь y на $\exp(-z/2^{27})$ 136 ⟩ Используется в разделе 135.
⟨ Умножь, когда хотя бы один операнд известен 942 ⟩ Используется в разделе 941.
⟨ Уничтожь полностью пустой символ 1168 ⟩ Используется в разделе 1167.
⟨ Упрости все существующие зависимости, заменяя на x 614 ⟩ Используется в разделе 610.

⟨ Установ параметры, нужные для *paint_row*; но **goto** *done*, если закраска не нужна в конце 582 ⟩
Используется в разделе 578.
⟨ Установи вызов макроса без суфикса и **goto** *restart* 853 ⟩ Используется в разделе 845.
⟨ Установи вызов макроса с суфиксом и **goto** *restart* 854 ⟩ Используется в разделе 852.
⟨ Установи комплексный множитель, затем **goto** *done* 959 ⟩ Используется в разделе 957.
⟨ Установи локальные переменные *x1*, *x2*, *x3* и *y1*, *y2*, *y3* на кратные управляющих точек повёрнутых производных 543 ⟩ Используется в разделе 541.
⟨ Установи начальные значения ключевых переменных 21, 22, 23, 69, 72, 75, 92, 98, 131, 138, 179, 191, 199, 202, 231, 251, 396, 428, 449, 456, 462, 570, 573, 593, 739, 753, 776, 797, 822, 1078, 1085, 1097, 1150, 1153, 1184 ⟩
Используется в разделе 4.
⟨ Установи отбраковываемые веса или **goto** *not_found*, если пороги плохи 1075 ⟩
Используется в разделе 1074.
⟨ Установи переменную q на узел в конце текущего октанта 466 ⟩ Используется в разделах 465, 506 и 506.
⟨ Установи переменную z на аргумент пары (x, y) 142 ⟩ Используется в разделе 139.
⟨ Установи переменные (*del1*, *del2*, *del3*), чтобы представить $x' - y'$ 421 ⟩ Используется в разделе 420.
⟨ Установи равенство для данного значения θ_0 293 ⟩ Используется в разделе 285.
⟨ Установи равенство для загиба в θ_0 294 ⟩ Используется в разделе 285.
⟨ Установи равенство для загиба в θ_n и **goto** *found* 295 ⟩ Используется в разделе 284.
⟨ Установи синусы и косинусы, затем **goto** *done* 958 ⟩ Используется в разделе 957.
⟨ Установи текущее выражение на желаемые координаты пути 987 ⟩ Используется в разделе 985.
⟨ Установи уравнение соответствующим фиктивным изгибам в z_k; затем **goto** *found* с θ_n, приравненным θ_0, если цикл закончился 287 ⟩ Используется в разделе 284.
⟨ Установи явные натяжения 882 ⟩ Используется в разделе 881.
⟨ Установи явные управляющие точки 884 ⟩ Используется в разделе 881.
⟨ Читай одну строку, но верни *false*, если место в памяти строк слишком мало 52 ⟩
Используется в разделе 51.
⟨ Читай остальные строки из файла `MF.POOL` и верни *true*, или дай ссобщение об ошибке и верни *false* 51 ⟩ Используется в разделе 47.
⟨ Читай первую строку нового файла 794 ⟩ Используется в разделе 793.
⟨ Читай следующую строку файла в *buffer* или **goto** *restart*, если файл закончился 681 ⟩
Используется в разделе 679.
⟨ Читай строку из терминала 897 ⟩ Используется в разделе 895.